《通信顺序进程》译注

原著[1]：C. A. R. Hoare

译注[2]： Michael H. Ji

[1] 1985 年 1 月 1 日印刷版，Prentice/Hall 和 2015 年 5 月 18 日电子版。
[2] 2018 年 3 月 11 日初译稿；2021 年 1 月 14 日修改稿。

Communicating Sequential Processes

C. A. R. Hoare

May 18, 2015

【此译注工作为非盈利性的教育目的。译者放弃和不拥有任何版权。可以被随意下载，转发和打印用于任何以非盈利性质的教育和科研的目的使用。如果需要更多的关于版权方面的了解，请参阅 1985 年 Prentice Hall 出版的版权声明和 2015 年牛津大学的 CSP 电子版的版权声明。】

【This translation and annotation work is for non-profit educational purposes. The author waives and does not own any copyright for the work herein. All the documents can be downloaded, forwarded and printed freely for any non-profit educational and scientific purposes. If you need more knowledge about copyright, please refer to the copyright notice published by Prentice Hall in 1985 and the electronic copyright notice of the CSP of Oxford University in 2015.】

谨以此译注纪念

数理逻辑与并发系统研究领域的先贤科学家们！

译者序

译者一直好奇这个复杂的世界是基于逻辑的，还是基于数学的。并发行为无处不在，大到浩瀚星宇，小到微观粒子。其共性是一个分布并发复杂系统(Complex System)，各个组成个体(Agent)独立的运行着，但同时彼此交换信息，彼此影响。复杂系统蕴含的巨大的不确定性，这使得人们对其本质的理解和把握充满了好奇。

感谢图灵奖获得者，英国皇家学会院士 Tony Hoare 关 CSP(Communicating Sequential Process)的开创性的工作。CSP 通过引入代数的手段，利用符号逻辑的方法来形式化描述和精确推演一个复杂系统的迭代行为，令人无比倾佩。

感谢中国科学院周巢尘院士上个世纪 80 年代与 Tony Hoare 之间的学术合作研究， 和在 1988 年领衔翻译出版了 Hoare 的这一经典著作，对中国计算机科学在数理逻辑，程序语言，软件可靠性验证的研究做出了巨大的贡献。我们永远心存感激。

未来的世界，万物互联。随着人工智能科技的兴起， 无人驾驶汽车、无人飞行器、智能机器人和海量物联网设备等等正在迅速的普及。这些大规模互联智能系统的安全性(Safeness) 、可靠性(Reliability) 、鲁棒性(Robustness) 、弹性(Resilience)和高可用性(Availability)变得对社会，家庭和个人息息相关(Mission Critical)，已经成为人类未来可持续发展的重要课题。

译者认为我们可能需要更多的从并发系统的本质来理解系统的不确定性， 从并发系统状态空间的爆炸性的角度来理解和设计未来的智能系统。这是为什么时隔 30 年，译者在 Hoare 的 1985 年出版的《Communicating Sequential Processes》[3]一书，牛津大学 Jim Davies 于 2015 年发布的 CSP 电子版本[4]，和周巢尘院士于 1988 年由北京大学出版社出版的《通信顺序进程》[5]一书工作的基础上，重新翻译和着重加注这本关于并发理论的经典著作，并反馈给社区的主要目的。

译者在并发理论方面水平有限，但对系统行为的不确定性，对系统行为的复杂性充满了好奇。希望这一非营利性的工作对社会，对相关的研究人员，学生有益。

---

[3] Hoare, C. A. R. [1985]. Communicating Sequential Processes. Prentice Hall International. ISBN 0-13-153271-5.

[4] http://www.usingcsp.com/cspbook.pdf.

[5] 周巢尘 [1988]. 通信顺序进程. 北京大学出版社. ISBN7-301-00813-9.

前言(FORWARD)

因为若干原因，所有知情的人都在焦急的等待着这本书的出版；他们的耐心现在终于得到了回报。

一个简单的原因是因为这本书是 Tony Hoare 的第一本专著。很多人是 Hoare 在世界各地不知疲倦地作学术讲演时认识他的；更多的人知道 Hoare 是因为他是不少学术文章的作者。其论文清晰、严谨，而且涉及面很广泛。他的文章通常还墨迹未干时，就已经成了相关领域的经典文献。但是，专著是一种非常不同于学术研究文章的媒介：在专著中作者可以不受通常十分严格的篇幅限制来表达自己的观点；作者可以获得更方便地表达自己，和对一些话题展开充分讨论的机会。Tony Hoare 充分的利用了专著的这些优点。不负众望。

另外一个更具体的原因是来自这本书的内容本身。作为一门新型学科，当计算机界在 25 年前开始认识到并发现象时，整个计算机界都陷入了无穷无尽的困惑之中。这种困惑一方面来自并发现象会出现在许多不同的各类环境中，另一方面是由于并发现象同时导致了许多当代历史事件发生的非确定性。为了解开这一困惑，需要一个成熟的、具有献身精神的而且是幸运的科学家的艰苦劳动来诠释这一复杂的并发理论。Tony Hoare 把他科学生涯中的一段主要精力用来了接受这一挑战，我们从各个角度而言都要感谢他。

期待这本书早日出版的最深刻的原因是读过本书早期初稿的人都有的一种强烈感受。Hoare 的书稿通过令人惊奇的清晰透彻的诠释并发理论，展现了计算机科学可以或者甚至说，应该成为怎样的一门科学。计算机科学家口头说或者自我感觉他们面临的主要挑战是不要被自己研究工作的复杂性弄糊涂了是一回事；而发现并通过一些数学法则的准确无误和优雅来达到这一崇高的目标，则是完全不同的另一回事。在这里，我们从 Charles Antony Richard Hoare 的科学智慧，数学符号系统的大胆使用，以及处理上的巧妙中受益匪浅。我们对此深深的心怀感谢。

Edsger W. Dijkstra[6]

[6] Edsger Dijkstra，1972 年图灵奖获得者。是结构化程序设计，并发控制理论，图算法等领域的先驱，是 20 世纪最伟大的计算机科学家之一。2002 年 8 月 6 日去世于荷兰。

序言(PREFACE)

本书是写给那些有抱负的，有志于对他们所从事的需要高智能的工作有更深入的理解和掌握更高技能需求的程序员的。本书的组织是通过对一个熟悉的话题采用一种新的探索方法，从而引起读者产生一种很自然的好奇心。这种新方法是通过一组例子来阐述的，这组例子选自一些不同的应用领域，从自动售货机、童话故事，游戏，到计算机操作系统。对这些例子的处理是通过一套系统化的数学代数理论来刻画的。本书的最根本的目的是让读者具有一种洞察力，能用新的眼光去考察当前和未来的问题，使这些问题能更有效、更可靠地得到解决，甚至在某些情况下，最好是可以避开从不需要考虑这些问题。

本书中阐述的新方法最显而易见的应用场景是可以用在形式描述、设计和实现那些持续不断与环境发生交互作用的计算机系统。基本的思路是这种系统可以很容易地分解成多个并行运行的子系统。这些子系统之间以及与它们的共同环境之间持续地发生交互作用。子系统的并行组合与传统程序设计语言中程序行或程序语句的顺序组合是一样的简单明了。

对并发系统的这种洞察理解也带来许多实际的好处。首先，可避免程序设计中处理并行性时的很多传统问题，如干扰、互斥、中断、多线程、信号量等等。其次，近年来在程序设计语言和程序设计方法学研究中提出的很多高级结构化观念，也都成为本书理论观点下的特例了，例如管程(Monitor)、类程(Class)、模块(Module)、包(Package)、临界区(Critical Region)、封装(Envelop)、形(Form)、以及最普通的子程序(SubRoutine)概念。最后，这种新方法还提供了可以避免严重错误，诸如并发程序的发散、死锁和非终止等，和计算机系统在设计和实现时的正确性可以得到严格保证的可靠的数学基础。

在书的组织安排上，我试图按照逻辑的和循序渐进的顺序逐步介绍我的关于并发理论的想法，先从简单和基本的算子开始，逐渐展向利用这些基本算子的更精巧的应用。好学勤奋的读者可以逐页地阅读本书。但其他读者可以从他们自身比较有兴趣的话题开始；为此本书的每一章都刻意组织，方便读者可以适宜的选读。

1. 每个新观念都给出一个非形式的说明，并用一些小例子来解释，这样可能有助于所有读者。

2. 书里给出了这些新概念的相关代数演算法则。这些法则可以用来刻描述各种运算的重要特性。喜爱数学优美结构的读者会对这方面有兴趣。如果读者希望通过能够保持正确性的变换来优化他们的系统设计，那么这些代数法则对他们也会非常有所帮助。
3. 书里使用了大家比较熟知的程序设计语言 LISP 的一个简单的功能子集来实现各个算子。这个略微不太一般。对于那些可以用 LISP 实现的手段来检验和论证其系统设计的读者，对本书的这一安排会很感兴趣。
4. 系统分析员对迹的定义和描述会应该感兴趣。系统分析员在开始一个系统的实现前，需要精确的描述用户的需求。高级程序员对这部分也应该有兴趣，因为在设计一个系统时，他们需要将一个系统划分为若干个子系统，并且需要清楚的描述子系统间的接口。
5. 证明规则对工程实现人员很有益。工程人员肩负着根据已有的描述、规定的日程、规定的价格，完成可靠的程序的重要责任。
6. 最后，书中的数学理论对进程概念和相关的用来构建各种进程的算子给出了严格的定义。这些定义是相关代数规则，实现和证明规则的基础。

读者对上述任一论题缺乏兴趣，或者感到一时很难理解，可以略去或者暂缓阅读相关章节。

各章间的承上启下也很方便于读者的精读、选读或者按照自己的安排。其中第一章和第二章的前几节是导引，建议所有的读者都阅读，但后面的章节可以略过，或者留待第二遍时再读。第三、四、五章是彼此独立的，读者可按兴趣和爱好，以任意次序或组合进行阅读。我们建议，读者在遇到难懂的内容时，不必中断，可继续阅读下一节，甚至下一章，因为忽略过的材料很可能不会立即又提到。如果需要用到前面的内容时，书中通常会标注好相关引用，从而当读者有了足够的动力时，可以重新捡起来。我希望读者最终能发现书中的每项内容都是有趣的，是值得一读的；但我并不期望每个读者都按本书书写的次序来阅读和掌握。

用来阐述本书中各种新概念的都是些小例子。这是有意这样安排的。解释一个新概念的最初的几个例子就应该十分简单。这样才不会由于例子的复杂或者读者的不熟悉而影响对概念的理解。通常后面的一些例子会比较难理解，因为这些例子涉及的问题本身很容易让人产生困惑而且很容易变得复杂化；由于本书引入的新概念的表达能力的强大，符号系统的精巧，从而对这些问题的解决方案会显得非常简洁。

然而，每个读者以后都会痛苦的遇到一些问题，这些问题会比书中的例子覆盖的范围更大，更复杂，更重要。这类问题看上去是用任何数学理论来处理都是很难的。但切不可因此而泄气或者放弃，应该接受挑战，试着应用本书中的新方法去解决这些现实的问题。

选择一个复杂问题的某些局部点，形成一个简化的模型，然后以此为出发点，在必要的时刻，再逐步加入复杂的成分。令我们惊奇的发现是，最初的、简化的模型往往会增加你对问题的深刻了解，有助于你对整个问题的解决。模型或许可作为一种基本结构，在这个结构上，复杂细节可以安全的地往上添加。令人最惊奇的发现是，在模型中添加的一些复杂细节是没有必要的。如果事情都是这样，掌握一种新的系统构建方法所作的努力是完全值得的，会有不错的回报。

在学习时，人们常常抱怨符号系统。例如，初学俄语的学生经常抱怨陌生的斯拉夫字母造成的障碍，特别是其中不少字母的发音奇特。尽管如此，这还是俄语学习中最容易的阶段。学会书写后，必须学习文法和词汇，然后掌握这一切，必须花时间去学，去练，去学会通过语言表达自己的观点。这一切都不会一蹴而成。学习数学也是这样。开始时，符号也是一道难逾的路障；但是实质性的问题是理解符号的含义和特性，了解如何使用它们，并且学会熟练地使用它们来描述新的问题、寻求解决方案、给出相关证明。最终，你能培养自己具有对数学优美风格的鉴赏力。当达到这一境界时，外在的符号会消失；通过符号你可直接看到其内在含义。数学的优点在于，它的规则比之自然语言的规则简单得多，它的词汇比之自然语言的词汇也少得多。因此，当你遇到一些陌生的数学问题时，你可以使用逻辑演绎和相关创造来解决这些问题，而不必请教书籍或专家。

这就是为什么数学会象程序设计那样有趣。可惜数学并不总是很容易。即使是数学家们在学习研究一个新的专业分支时，也会觉得很困难。通信进程(CSP)的理论是数学的一个新分支；因此，学习通信进程时，程序人员与数学家相比，并不处于劣势。 但在程序人员最后在把获得的知识付诸实践时，会有明显的优势。

本书的内容在一些非正式的会议上和正式的学校课程教学上讲过。本书曾用于软件工程专业的硕士课程，讲授一学期。但大部分材料也可用作计算机系的本科最后一年或者大学第二年的课程来讲授。要求的预备知识主要是高中代数、集合论概念、谓词演算符号等。这些都列于序言后的汇总的符号表中。这本书的内容也可以为有经验的程序人员举办为时一周的高强度的集中培训课程。在讲授时，教师应重点讲解例

子和定义，而将更多的数学成分留作课后自学。如果时间确实稀少，课时少于一周时，即使只能讲完第二章也是值得的；如果仔细选材的话，还可举办一小时的讨论班，讲到很有启发意义的五个哲学家就餐的故事。

讲授通信顺序进程是十分有趣的事情，因为书中的例子提供讲演者发挥演剧才能的机会。一个例子相当一个剧本，演出时一般着重表演有关人物的感情。听众通常会从演出中感到死锁是特别滑稽的。但是听众要始终警惕这种人格化的危险性。讲演者籍助表演动机、爱憎、感情波动，“使枯燥、荒诞的故事听起来逼真”，而数学公式有意摆脱了这些人为因素。大家应该专心致志于理解数学公式的无人情味的、干巴巴的含义，并且学会欣赏数学抽象的优美。比如，某些递归定义的算法就具有 J.S.Bach 所创作的逃亡曲般的惊人的优美[7]。

[7] 这一段文字充分借鉴了周巢尘院士的 1988 年版本的翻译。

概要(SUMMARY)

第一章介绍进程的基本概念[8]，进程是系统及其环境间交互作用的一种数学抽象。我们熟悉的递归技术可用来刻画一个生命周期很长的，或者永不消亡的进程。该章中的概念先通过例子和图形来解释；更完整的解释是由代数法则和用函数式程序设计语言的实现来给出。该章第二部分阐述一个进程的行为可由进程所从事的一系列动作的轨迹(Trace)来记载。我们定义了许多关于迹的代数和逻辑运算。在实现一个进程前，可以利用这个进程的迹的各种特性来描述它。该章中给出了一些规则，从而可以帮助确保一个进程的实现与其形式描述是满足一致性的。

第二章阐述了如何将多个进程组装成系统，在系统中各子部件交互作用，也和系统的外部环境彼此打交道。引入并发性本身并不产生任何非确定性[9]。这一章中的主要例子是经典的五个哲学家的就餐问题，本章第二部分中阐述了，如何利用改变进程能够参与的事件的名字的方法，使得进程可以很方便地具有新用途。这章最后以确定性进程的数学理论为结束，其中包括递归理论的一个简单介绍。

第三章对令人困惑的非确定性问题给出一个最简单的解。非确定性是实现抽象的一种重要技术。当我们决定忽略或者封装掩盖一个系统行为中我们不感兴趣的部分时，就会自然地会出现非确定性。非确定性也具备对称性，这种对称性可见相关算子定义的数学理论部分。非确定性进程的证明方法比确定性进程稍微麻烦些，因为必须证明每一种可能的非确定选择产生的行为都符合进程的描述。幸运的是，我们在本书中提出了一些绕过非确定性的技术，这些技术会在第四和第五章中被广泛的使用。因此，对于第三章的学习或精通掌握可一直推迟到第六章。第六章中就不能再避免引入非确定性了。

第三章的后面一些章节中，给出了非确定性进程概念的一个完整的数学定义。希望能够探究这门学科的理论基础，或者希望用证明的方法来检验书中的代数法则和其它进程特性的有效性的纯数学家们应该对这个定义会有兴趣。而应用数学家(包括程

[8] Tony Hoare 的原著术语是"Process"。周巢尘院士在 1988 年译本翻译为“进程”。本书在符号系统和概念命名上大多数沿用《通信顺序进程>>(周巢尘 1988 北京大学出版社)一书。译者认为“Process”如果翻译成“过程”可能略微贴切一些，例如，控制系统中的“过程控制”。读者不要与操作系统中的进程与并发理论 CSP 中的进程混淆。操作系统中的进程指的是一个程序的实例(Instance)。

[9] 在 CSP 里，Hoare 对并发问题的处理引入的主要是如何诠释同步问题，不考虑非确定性问题。非确定性问题是通过一个单独的算子来表达。

序设计人员)可能认为这些法则是不证自明的，或者通过这些法则的实用性来证明其有效性，这些读者可以不需要去阅读那些比较理论的章节。

第四章开始介绍通信：通信是两个进程间相互作用的一种特例，其中的一个进程输出一个消息，与此同时，另一个进程输入这个消息。因此，通信是同步的；如果要在通道上使用消息缓冲，可在两个进程间插入一个缓存进程。

设计并发系统的重要目的是在解决实际问题时获得高性能计算。为了阐述这一目的，书中设计了一些简单的脉动式(或迭代式)算法。管道[10]就是一个简单例子。管道定义为由一系列进程所组成，其中每一个进程仅从它的先行进程输入消息，仅向它的后继进程输出消息。管道在实现单向的，具有层次结构的通信协议时很有用。最后，在本章中我们定义了一个重要的抽象数据类型，从属进程。从属进程的每个进程实例只和其从属于的母进程通信[11]。

第五章讲述如何在通信顺序进程的框架内集成传统的顺序程序设计的算子。有经验的程序人员可能会很吃惊的发现，这些程序设计语言算子和常用的数学理论中的算子一样，具有相同漂亮的代数性质；另外，证明顺序程序满足其规约描述，和证明并发程序满足规约描述的方法非常想象。即使外部启动的中断也可用通信顺序进程定义，该章中会说明其用途。中断也遵从通信顺序进程理论的相关法则。

第六章讨论怎样构造和实现一个复杂系统，在这个系统中有很多的进程，它们共享有限的物理资源，诸如磁盘、行式打印机等，而且进程对资源的需求是随着时间而变化的。每一个资源由一个进程来表示。每当用户进程需要一个资源时，就建立一个新的虚拟资源。一个虚拟资源是一个进程，它有点象用户进程的附属进程；但可以和实际第物理资源通信。这些通信是和其它同时活动着的虚拟进程的通信交叉进行的。这里的实际和虚拟进程概念与 PASCAL PLUS 中的管程(Monitors)和闭体(Envelopes)扮演的角色是一样的。这一章通过用模块式开发一系列完整但是非常简单的操作系统来解释如何构造一个复杂系统。这些例子堪称本书中规模最大的例子。

第七章阐述了研究并发和通信的其他方法，并且解释了导致本书前面章节中理论的技术、历史和作者个人的动因。在本章中，我深深的感谢其他作者对我的影响，并推荐和介绍这个领域中可以进一步阅读的材料。

---

[10] 周巢尘院士的译本中为“导管”。译者认为“管道”应该更为贴切，符号现代操作系统的概念和机制。

[11] 原文用了“Subordinate Process"，从属进程。读者可以直观的理解就是类似程序语言中的子程序的概念。

致谢(ACKNOWLEDGEMENTS)

兹深深的感谢 Robin Milner 先生深刻的、原创性的研究工作。其研究工作在他的开山之作通信系统演算(Calculus for Communicating Systems)中有详尽的说明。他独有的洞察力，他的友谊和他在学问上对我的的鞭策，一直是本书竟成的各项工作的灵感和勇气的源泉[12]。

在过去 20 年中，我一直思考着并行计算程序设计中的一些问题，并想设计一种程序设计语言以缓解这些问题。这段时间中，我极大得益于与很多科学家的合作，包括 Per Brinch Hansen, StePhen Brookes, Dave Bustard, Zhou Chao Chen[13], Ole-Johan Dahl, Edsger W. Dijkstra, John El-der, Jeremy Jacob, Ian Hayes, Jim Kaubisch, John Kennaway, T. Y. Kong, Peter Lauer, Mike McKeag, Carroll Morgan, Ernst-Rudiger Olderog, Rudi Reinecke, Bill Roscoe, Alex Teruel, Alastair Tocher 和 Jim Welsh。

最后，特别感谢 O.-J. Dahl, E. W. Dijkstra, Leslie M. Goldschlager 和 Jeff Sanders 等人，他们仔细阅读了本书的初稿，并且指出了原稿中错误和费解处；也要特别感谢一九八三年一月参加了 Wollongong 暑期计算机程序设计科学讲座的参加者，一九八三年四月在中国科学院研究生院参加了我的讲习班的人们，以及一九七九年至一九八四年牛津大学计算专业的硕士生们。

[12] Robin Milner(13 January 1934 – 20 March 2010)，1991 年图灵奖获得者，英国爱丁堡大学计算机教授和系主任。英国爱丁堡皇家院士，英国皇家学会院士，美国 ACM 院士，美国工程院外籍院士。著名的爱丁堡大学计算机系 LFCS(Laboratory for Foundations of Computer Science)的创办人之一。 是机器自动定理证明，函数程序语言，并发系统分析等领域的开创人物。其 CCS 的工作与 Hoare 的 CSP 基本上属于同一个时期的，分别独立的原创性研究工作。

[13] 中国科学院院士周巢尘院士。1958 年毕业于北京大学数学力学系。1967 年研究生毕业于中国科学院计算技术研究所。中国科学院软件研究所研究员，联合国大学国际软件技术研究所所长。研究生期间，研读数理逻辑，师从中国数理逻辑研究领域科学家胡世华院士。

符号表(GLOSSARY OF SYMBOL)

逻辑符号(Logic)

记号	含义	例子
=	相等	x=x
≠	不等	x≠ x+1
□	例子或证明的结束符	
P∧Q	P 和 Q(两者为真)	x≤x+1∧x≠x+1
P∨Q	P 或 Q(两者或其一为真)	x≤y ∨ y≤x
¬ P	非 P(P 不真)	¬ 3>5
P=>Q	若 P 则 Q	x<y => x≤y
P≡Q	P 当且仅当 Q	x<y ≡ y>x
∃ x.P	存在 x 使 P 真	∃ x.x＞y
∀x.P	对一切 x,P 真	∀x.x＜x+1
∃ x：A.P	存在集合 A 中元素 x,使 P 真	
∃ x：A.P	对集合 A 中一切元素 x,P 真	

集合(Sets)

记号	含义	例子
∈	属于	拿破仑 ∈ 人类
∉	不属于	拿破仑∉俄罗斯人
{ }	空集(无元素集)	¬(拿破仑 ∈{})
{a}	a 组成的单元素集 a 是其仅有的元素	x∈{a} ≡ x=a
{a,b,c}	a,b,c 组成的集	c∈{a,b,c}
{x | P(x)}	使 P(x)为真的全体 x	{a}={x | x=a}

$A \cup B$	A 并以 B	$A \cup B=\{x \mid x \in A \vee x \in B\}$
$A \cap B$	A 交以 B	$A \cap B=\{x \mid x \in A \wedge x \in B\}$
$A - B$	A 减去 B	$A - B=\{x \mid x \in A \wedge \neg x \in B\}$
$A \subseteq B$	A 包含于 B	$A \subseteq B \equiv \forall x：A.x \in B$
$A \supseteq B$	A 包含 B	$A \supseteq B \equiv B \subseteq A$
$\{ x：A \mid P(x) \}$	使 P(x)真的 A 中全体 x	
$\mathbb{N}$	自然数集	{0,1,2...}
$\mathbb{P}A$	A 的幂集	$\mathbb{P}A=\{X \mid X \subseteq A\}$
$\cup_{n \geq 0} A_n$	集合族的并集	$\cup_{n \geq 0} A_n = \{x \mid \exists\ n \geq 0 . x \in A_n\}$
$\cap_{n \geq 0} A_n$	集合族的交集	$\cap_{n \geq 0} A_n = \{x \mid \forall n \geq 0 . x \in A_n\}$

函数(Functions)

记号	含义	例子
f：$a \rightarrow b$	f 是将 A 中每个元素映射到 B 中元素的一个函数	square：$\mathbb{N} \rightarrow \mathbb{N}$
$f(x)$	(A 中)x 经过 f 得到的 B 中的映象	
单射	将 A 中元素映射到 B 中不同元素的函数	$x \neq y \Rightarrow f(x) \neq f(y)$
f^{-1}	单射 f 的逆	$x=f(y) \equiv y=f^{-1}(x)$
$\{f(x) \mid P(x)\}$	将 f 作用于使 P 为真的全体 x 所得到的集合	
$f(C)$	由 f 形成的 C 的映象集	$\{y \mid \exists x . y = f(x) \wedge x \in C\}$ square({3,5})={9,25}
$f \circ g$	f 复合 g	$f \circ g(x) = f(g(x))$
$\lambda x . f(x)$	将 x 的每个值映射至 $f(x)$的函数	$(\lambda x . f(x))(3) = f(3)$

迹(Traces)

节号	记号	含义	例子

1.5	<>	空迹	
1.5	<a>	仅含 a 的迹	
		(单元序列)	
1.5	<a,b,c>	由 a 然后 b 然后 c 构成的迹	
1.6.1	^	(迹间的)相接	<a,b,c> = <a,b> ^ <> ^ <c>
1.6.1	s^n	重复 s 共 n 次	$<a,b>^2$ = <a,b,a,b>
1.6.2	s↾A	s 受限于 A	<b,c,d,a>↾{a,c} = <c,a>
1.6.5	s≤t	s 是 t 的前缀	<a,b> ≤ <a.b.c>
4.2.2	$s \leq^n t$	从 t 的尾部至多移走 n	$s \leq^n t$
		个符号后可得到 s	
1.6.5	s in t	s 在 t 中	<c,d> in <b,c,d,a,b>
1.6.6	#s	s 的长度	#<b,c,b,a> = 4
1.6.6	s↓b	s 中 b 的个数	<b,c,b,a>↓b ＝4
1.9.6	s↓c	s 中记载的通道的通信	<c.1,a.4,c.3,d.1>↓c = <1,3>
1.9.2	^/s	压扁后的 s	^/<<a,b>,<>,<c>> = <a,b,c>
1.9.7	s;t	s 成功地接以 t	(s^<√>);t = s^t
1.6.4	A*	A 中元素的序列集	A*={s|s↾A=s}
1.6.3	s_0	s 的首元素	$<a,b,c>_0$ = a
1.6.3	s´	s 的尾部	<a,b,c>´ = <b,c>
1.9.4	s[i]	s 的第 i 个元素	<a,b,c>[1] = b
1.9.1	*f**(s)	s 的 *f* 星函数	square*(<1,5,3>)=<1,25,9>
1.9.5	$\overline{s}$	s 的逆置	$\overline{< a,b,c >}$ = <c, b, a>

特殊事件(Special Events)

节号	记号	含义
1.9.7	√	成功(成功终止)
2.6.2	τ.a	名为 τ 的进程参予事件 a
4.1	c.v	在通道 c 上传递值 v
4.5	τ.c	在通道 τ.c 上传递消息 v

4.5	τ.c.v	名为τ的进程的通道 c
5.4.1	↯	灾难(闪电)
5.4.3	⊗	交换
5.4.4	©	为恢复所设的检查点
6.2	acquire	获取
6.2	release	释放

进程(Processes)

节号	记号	含义
1.1	αP	进程 P 的字母表
4.1	αc	在通道 c 上可传递的消息的集合
1.1.1	a → P	a 然后 P
1.1.3	(a → P | b → Q)	在 a 然后 P 与 b 然后 Q 间的选择(如果 a≠b)
1.1.3	(x: a → P(x))	在 A 中选取 x 然后 P(x)
1.1.2	μX: A.F(X)	满足 X=F(X)的字母表为 A 的进程 X
1.8.3	P/s	执行迹 s 中事件后的 P
2.3	P ‖ Q	P 和 Q 并行执行
2.6.2	l ： P	P 冠以名 l
2.6.4	L:P	P 冠以集合 L 中的名字
3.2	P ⊓ Q	P 或 Q(非确定性的)
3.3	P□Q	P 和 Q 间的选择
3.5	P\C	除去 C 后的 P(屏蔽)
3.6	P|||C	P 和 Q 的交叉
4.4	P >>Q	P 链接以 Q
4.5	P // Q	P 附属于 Q
6.4	l :: p // Q	远程从属
5.1	P;Q	P(成功地)随之以 Q
5.4	P△Q	P 被 Q 中断
5.4.1	P↯Q	P 在遇到灾难后执行 Q

5.4.2	$\hat{p}$	可再启动的 P
5.4.3	P⊗Q	P 和 Q 交替执行
5.5	P◁b▷Q	若 b 则 P，否则 Q
5.1	*P	重复执行 P
5.5	b*P	若 b 则重复 P
5.5	x：=e	x 取(值)e
4.2	b!e	在(通道)b 上输出(值)e
4.2	b?X	从(通道)b 上向 x 输入
6.2	l ! e?x	调用名为 l 的共享子程序，值参为 e，结果送至 x
1.10.1	P sat S	(进程)P 满足(规约)S
1.10.1	tr	给定进程的任意迹
3.7	ref	给定进程的任意拒绝集
5.5.2	$x^{\surd}$	给定进程所产生的 x 的终止值
5.5.1	var(P)	可由 P 赋值的变量集
5.5.1	acc(P)	可由 P 引起的变量集
2.8.2	P⊑Q	(确定性地)Q 能完成 P 能做的事情
3.9	P⊑Q	(非确定性地)Q 不比 P 差
5.5.1	D e	表达式 e 有定义

代数(Algebra)

术语	含义
自反(reflexive)	满足 xRx 的关系 R
反对称(antisymmetric)	满足 xRy ∧ yRx ⇒ x=y 的关系 R
传递(transitive)	满足 xRy ∧ yRz ⇒ xRz 的关系 R
偏序(partial order)	自反、反对称和传递的关系≤
底(bottom)	满足⊥≤x 的最小元素⊥
单调(monotonic)	保持偏序的函数 f，即 x≤y ⇒f(x) ≤ f(y)
严格(strict)	保持底值的函数 f，即 f(⊥)=⊥
幂等(idempotent)	满足 x f x = x 的二元算子 f

对称(symmetric)	满足 x *f* x = y *f* x 的二元算子 *f*
结合(associative)	满足 x *f* (y *f* z) = (x *f* y) *f* z 的二元算子 *f*
分配(distributive)	*f* 可分配入 g，若 x *f* (y g z) = (x *f* y) g (x *f* z)
	而且 (y g z) *f* x = (y *f* x) g (z *f* x)
单元位(unit)	满足 x *f* 1 = 1 *f* x = x 的元素 1，是 *f* 的单位元
零元(zero)	满足 x *f* 0 = 0 *f* x = 0 的元素 0.是 *f* 的零元

图(Graphs)

术语	含义
图(graph)	通过结点和边的关系的一种结构的表示方法
节点(node)	图的一个节点，表示该关系的定义域或值域中的一个元素
弧(arc)	图的一线段或带有箭头的线段，连接满足该关系的两个节点
无向图(undirected graph)	对称关系的图
有向图(directed graph)	非对称关系的图，图中的弧通常是有向的箭头
有向回路(directed cycle)	一组方向相同的箭头组成的回路的节点集
无向回路(undirected cycle)	一组方向不定的弧或箭头组成的回路的节点集

目录(Contents)

第一章　进程(PROCESSES)

1.1 引言(Introduction)

让我们暂时把计算机和计算机程序设计忘掉一会儿，想想我们周围世界的各种对象[14]，这些对象依据各自的某种行为方式在活动，并与我们或其它对象发生相互作用。这类对象如钟表、筹码机、电话机、博弈游戏和自动售货机等。要描述它们的行为，我们首先要确定我们对这些对象的哪类事件或动作感兴趣，然后给每类事件赋予一个不同的名称。就拿一个简单的自动售货机来说，可有两类事件：

coin — 将一枚硬币投入自动售货机的硬币槽；

choc — 从机器的发货器送出一块巧克力。

而对较为复杂的自动售货机来说，就有更多种类的事件(event)[15]：

in1p — 投入一枚一便士的硬币；

in2p — 投入一枚两便士的硬币；

small — 送出一块小饼干或小甜饼；

large — 送出一块大饼干或大甜饼；

out1p — 送出一便士的找头。

注意每个事件名代表的是一个事件的类别(Class)；同一类的事件会在不同的时间场合出现很多次[16]。事件类别和具体出现的一个事件之间的区别与字母 h 的情形类似。同一字母 h 在本书中可以多次出现，只是出现在不同的地方而已。

我们把认为是与描述一个对象行为有关的全体事件的名称的集合叫做这个对象的字母表(alphabet)。字母表是一个对象的一个预先规定好的永久属性。从逻辑上讲，对象不可能执行其字母表以外的事件；譬如说，专售巧克力的售货机不可能突然间送出一艘玩具军舰来。但这个命题的逆命题并不成立。对象字母表内的事件也不一定会发生[17]。例如，专售巧克力的售货机可能就真的不能再吐出一个巧克力了——也许是因为售货机里还没装巧克力，也许是售货机坏了，也许是没人买巧克力了。但一经确定

---

[14] 原文是 Objects。泛指一个对象，可以是一个物理的实体，或者一个虚拟的实体。但在 CSP 中，读者不要与 Object Oriented Programming 的 Object 混淆。在 OO 中，Object 是 Class 的一个 Instance。在本书中，Object(对象)有点 Class 的含义，泛指物体，实体。

[15] 一个事件(event)通常意味着一个动作(action)；但一个动作可以是多个原子事件的组合。

[16] 类似一个事件(Class)可以出现许多实例(Object Instance)

[17] 或者永远不发生。只是属于行为集合的一部分。可能发生，但不一定能够触发。类似每个人都可能做总统，但要看运气。

choc事件是在自动售货机字母表内，即使该事件从来没有发生过[18]，也仍然被认为是属于自动售货机的事件。

选择一个对象的字母表一般要涉及到一点有意简化的问题，例如，舍弃那些我们不感兴趣的属性和动作。例如，我们不需要刻画自动售货机的颜色、重量和形状。另外，我们也有意舍弃某些即使与售货机本身密切相关的事件，如补充巧克力或倒空硬币盒——因为这些事件与顾客没关系(也不应有什么关系)。

在一个对象的生命周期中，每一个实际发生的事件应被看作是无延时的瞬息动作或原子动作。对于一个延续的或占时间的动作需要通过两个单独的事件来表示，一事件表示动作的开始，另一事件表示动作的结束；从动作的开始事件的发生到动作的结束事件的发生，有一段间隔，动作的延时就由这段间隔来代表；在这段间隔内，还可能发生其它事件。如果有两个延续动作，前一动作的还未结束，后一动作就开始了，则它们在时间上发生重叠。

我们有意忽略的另一个细节是事件发生的具体时间。这样做的好处在于简化了事件的设计和论证过程，而且可应用于具有任何运算速度的性能的物理计算系统。在那些对响应时间至关重要的应用场景下，需要单独考虑这些实时性问题。这些考虑是独立于设计逻辑正确性方面的。一个高级程序设计语言是否成功的必要条件就是与时序无关[19]。

忽略了事件的时间这个问题的好处是，我们不需要回答，也不需要问某一事件是否与另一事件丝毫不差地同时发生这些问题。当两个事件发生的同时性很重要时(如同步情形)，我们就把它们当做是单个事件的发生；不重要时，我们就将两个可能同时发生的事件按任意前后次序记录下来。

选择字母表时，没有必要区别由对象引发的事件(如 choc，吐出一个巧克力)和由对象外部的某个因素引发的事件(如 coin，购物者主动投币)。在 CSP 事件处理时，避开因果的概念能大大简化 CSP 理论及其应用。

从现在开始，我们用进程(Process)这个词来代表对象的行为模型，并认为一个进程的行为可以通过由组成其字母表的有限事件的集合来刻画。本书中，我们遵循如下约定：

---

[18] 原文是"that event never actually occurs"，意味着这个事件可以从来没有发生过。而非"不再发生"。

[19] 从另外一个角度而言，CSP 是从编程语言的角度来探索并发问题，例如，通过并发算子的能力。CSP 比较关注的是并发程序的逻辑正确性问题，不涉及时序逻辑问题。

1. 用小写的字符串表示不同事件，如：

coin, choc, in2p, out1p

有时也用字母来表示事件，如：a, b, c, d, e

2. 用大写的字符串表示相关具体的进程，如：

VMS——简单自动售货机

VMC——复杂自动售货机

在后面法则中出现的字母 P,Q,R 表示任意进程。

3. 字母 x, y, z 是表示事件的变量。

4. 字母 A, B, C 表示事件集合。

5. 字母 X, Y 是进程变量。

6. 进程 P 的字母表记作 αP, 如：

αVMS= {coin, choc}

αVMC= {in1p, in2p, small, large, out1p}

以 A 为字母表，但从不实际执行[20]A 中事件的进程，叫做 $STOP_A$。$STOP_A$ 刻画了一个毁坏了的对象的行为：尽管这个对象具备了执行 A 中事件的物理能力，但它从不执行这些能力。值得注意的是，字母表不同的对象，即使不做任何事情，它们也是不同的。例如，STOPαVMS 的行为是本可以吐出一块巧克力的。而 STOPαVMC 却永远不可能给出一块巧克力来，只能给出饼干。顾客即使根本不知道这两台自动售货机都坏了，也知道上述事实[21]。

在引言的余下部分中，我们要定义一些简单的符号约定，以帮助我们来表述能够实际做一些事情的对象。

1.1.1 前缀(Prefix)

设 x 为一事件，P 为一进程，则(x → P) (读做“x 然后 P”)。刻画了这样的一个对象的进程：它首先执行事件 x，然后按照进程 P 的说明进行动作[22]。我们定义进程(x →

[20] 或者从来不被外在的环境触发。

[21] 在第 5 章—顺序进程一章中，介绍了一个特殊进程 $SKIP_A$。与 STOP 进程代表一个处理过程的非正常不工作(例如，死锁(Deadlock)或者活锁(Livelock))不同的是，SKIP 进程代表一个处理过程的正常终止。如果不考虑一个 Process 的递归调用，一个 Process 在 CSP 的符号系统里必须要么是以 STOP 异常终止，或者 SKIP 成功结束。 读者可以参阅 5.1 节关于 SKIP 进程的定义。这个概念建议与 STOP 同时学习比较好。

[22] 是一种递归定义的方法。一个进程的行为，是通过另外的进程行为来表达。

P)与进程 P 有同样的字母表，所以，只有当 x 在 P 的字母表内时，这个记法才有意义[23]；形式地记为：

当 $x \in \alpha P$ 时， $\alpha(x \rightarrow P) = \alpha P$

例子

X1 一台简单自动售货机在损坏前接受了一枚硬币，记为：

$(coin \rightarrow STOP_{\alpha VMS})$ □

X2 一台简单自动售货机在损坏前成功地为两位顾客服务，记为：

$(coin \rightarrow (choc \rightarrow (coin \rightarrow (choc \rightarrow STOP_{\alpha VMS}))))$

最初，售货机只接受投入硬币槽内的一枚硬币，不允许先送出巧克力。当第一枚硬币投入后，硬币槽就关闭，直到一块巧克力被送出后再打开。这台售货机不会连续接受两枚硬币，也不会连续给出两块巧克力。 □

我们约定：→是一个右结合，这样以后我们可略去事件的线性序列中的括号，例如 X2 中的括号。

X3 一个筹码从一块板的左下方开始移动，只允许向上或向右移入相邻的空白方格内。

$\alpha CTR = \{up, right\}$

$CTR = (right \rightarrow up \rightarrow Right \rightarrow Right \rightarrow STOP_{\alpha CTR})$ □

注意算子→的右侧总是进程，左侧总是单个事件。如果 P 和 Q 为进程，则 P→ Q 在语法上是不正确的。某一进程首先按 P 进程动作，然后按 Q 进程动作，对这类进程的正确表述方法将在第四章中讲解。类似地，如果 x, y 为事件，从语法上讲，下述写法也是错误的[24]：$x \rightarrow y$。这类进程正确表述是：$x \rightarrow (y \rightarrow STOP)$。这样我们就将事件的概念和进程的概念区别开了，一个进程也许参与很多事件，也许什么也不做。

---

[23] (x → P)的含义是发生 x 事件发生后，一个处理过程(Process)下一步的处理过程(Process)。

[24] CSP 里进程的概念有点不精确。例如，如果 y 是第五章里定义的成功结束事件"√"，那么 x → y 其实应该等价于 x → SKIP 的行为。但在"→"的定义下，x → y 不是进程的语法，需要 x →√→ SKIP 才是表达了一个进程。

1.1.2 递归(Recursion)

对一个最终要结束的进程，我们可以用上述的前缀记法表述它的全部行为。但是如果将一台自动售货机在其预期的最长生存期内的全部行为都罗列出来，却是件极为无聊的事情；因此，我们需要有一种方法，用短得多的符号语法将重复的行为刻画出来。最好在使用这些语法时，不需要事先确定对象的寿命长短；这样，不受时间限制的不断动作并与其环境互相作用的对象，也就可以被表述了。

如果不考虑上弦的动作，钟是一种最简单的、有可能永远动作的对象，它只会滴答滴答地走，即

αCLOCK = {tick}

下面我们考虑一个对象，除一开始先走一下外，其行为与这座钟完全相同，可记为

(tick → CLOCK)

这个对象的行为是与现实中那座钟的行为是无法区分，等价的。所以，这一推理导出下列方程

CLOCK = ((x → P))

这个方程可以看作是钟的行为的一个隐性的定义，就象 2 的平方根可定义为下列方程中 x 的正数解一样

$x = x^2+x-2$

由闹钟的方程可得到一些不言而喻的结果，这些结果可以通过简单的等式代换得到。

CLOCK = (tick → CLOCK)　　原方程

= (tick → (tick → CLOCK)　　经过代换

CLOCK = (tick → tick → tick → CLOCK)　类似的持续代换得出

方程式可按需要进行多次展开，展开的次数不受限制。钟的潜在的无限行为可定义为

tick → tick → tick → ...

这就象 2 的平方根被认为是十进制数列

1.414... 的极限值。

这种通过自我引用的递归定义一个进程行为的方法，只有当方程式右边所有出现相应的递归重复调用的进程名之前，至少存在一个前缀事件时[25]，这种递归的进程定义方法才是有效的。例如，递归方程

$$X = X$$

不能说明任何问题，因为方程的解是任意的。我们把以前缀开始的进程表达式称作是卫式表达式(guarded)[26]。如果 F(X)是含进程名 X 的卫式表达式，A 为 X 的字母表，则我们断言方程

$$X = F(X)$$[27]

有唯一解。为方便起见，我们把这个解表示为

$$\mu X{:}A \bullet F(X)$$[28]

的形式。这里，X 是局部名(约束变量)，并且可以随意更换，即

$$\mu X{:}A \bullet F(X) = \mu Y{:}A \bullet F(Y)$$

这个等价性可通过方程

$$X = F(X)$$

中的变量 X 的解也是方程

$$Y = F(Y)$$

中变量 Y 的解来理解。

以后，我们或者用方程式，或用更加方便的 μ 表达方式给出进程的递归定义。在用 μX:A.F(X)定义时，如果字母表 A 从进程的内容或上下文中已经很清楚了，我们就略去不写。

例子

X1 一只永动钟，它的递归定义是

$$CLOCK = \mu X : \{tick\}.(tick \rightarrow X)$$ □

X2 一台简单自动售货机，投入多少枚硬币就送出多少块巧克力，其方程递归定义是

---

[25] 前缀(prefix)定义为(x → P)，其中 x 是一个事件，P 是一个进程。例如，(tick → CLOCK)的递归定义中，(tick → CLOCK)就是一个前缀：tick 是事件，CLOCK 是自我引用的递归进程的名字。

[26] 读者可参阅 Dijkstra 的经典论文"Guarded commands, nondeterminacy and formal derivation of programs", Communications of the ACM Volume 18 Issue 8, Aug. 1975 Pages 453-457.

[27] 这里的大写 X 是代表递归中的进程的符号(名)，例如，CLOCK，VMS 等。

[28] 卫式递归方程式定义为存在唯一解，而非类似 X = X，可以有多个解或者任意多个解。严格的数学证明会在后面给出。

VMS =(coin → (choc →VMS))

如上所述，这个方程式可以更形式化的通过 μ 的方式定义为：

VMS =μX : {coin, choc}.(coin → (choc → X))

这种 μ 的方式是方程式的另一种写法[29]。 □

X3 一台换钱机，投入五便士的硬币，它自动根据处理流程，换出零钱，表述为

αCH5A = {in5p, out2p, outlp}

CH5A = (in5p → out2p → outlp → out2p → CH5A) [30] □

X4 一台换钱方式不同的机器，其进程的字母表与上例相同，定义为

CH5B = (in5p → out1p → out1p → out1p → out2p → CH5B)[31] □

关于卫式的方程有解，且这个解是唯一的断言，可以通过字符串代换的方法非形式地证明。用方程的完整右侧对方程式里含有该进程自我引用的地方做一次代换，会导致定义进程行为的表达式随之增长，不断的持续替换会使得原始方程式所刻画的行为的初始部分的行为字符串越来越长。因此，通过这种替换的方法，任何有限长的行为都可以确定下来。在任一时刻动作都相同的两个对象具有同样的行为，例如，两个对象同属于同一个进程。目前觉得这种非形式的推理方法不好理解或不能令人信服的读者，可以暂时把这个结论当作公理加以接受，这条公理的价值和意义会越来越明显[32]。在没有给出进程的精确的数学定义前，严格的形式证明是无法给出的。这个定理的形式证明将在第 2.8.3 节中给出。在这里对递归的叙述主要以递归方程的卫式特性为基础。我们将在 3.8 节中讨论非卫式的递归的含义。

1.1.3 选择(Choice)

使用前缀和递归的方法，我们可以刻画那些只具备单一行为的对象，这类对象的行为不受外界影响。然而，有许多对象在其所处的环境中与环境相互作用，致使对象的行为受到环境影响。例如，一台自动售货机可能有两个硬币槽，一个只能投入二便士的硬币，另一个只能投入一便士的硬币；选择触发哪一事件是由顾客来决定的。假设 x 和 y 是两个不同事件，则表达式

---

[29] 通过 μ 的方式，一个递归进程可以通过 X 和 F(X)来表达，这样可以避免方程表达方式的束缚，然后可以把一个 μX.F(X)的表达作为一个符号系统的实体，去参加各种代数(Algebra)演算。这是 CSP 理论里一个非常重要的技巧从而可以把进程行为的研究转换为进程代数研究。

[30] CH5A 的代数形式定义为：CH5A =μX : {in5p, out2p, out1p}.(in5p → out2p → out1p → out2p → X)

[31] CH5B = μX : {in5p, out2p, out1p} • (in5p → out1p → out1p → out1p → out2p → X)

[32] 关于卫式的方程有解，且这个解是唯一的断言

$(x \rightarrow P \mid y \rightarrow Q)$

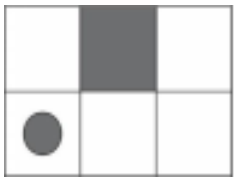
表述了这样一种对象，一开始时，它既可以执行事件 x，也可以执行事件 y。当第一个事件发生后，如果发生的是事件 x，则对象的后续行为按 P 进行动作；如果是事件 y，则对象按 Q 进行动作。因为 x 和 y 不同，则 P 和 Q 之间的选择就是由实际发生的第一个事件来确定。与以前一样，我们继续延用字母表的一致性，即有

$\alpha(x \rightarrow P \mid y \rightarrow Q) = \alpha P$　假设 $\{x, y\} \subseteq \alpha P$ 且 $\alpha P = \alpha Q$ 时[33]

"|"读做"选择"："选择 x 则按 P 执行，选择 y 则按 Q 执行"。

例子

X1 在下面这块板上筹码的移动可由下述进程所定义



$(up \rightarrow STOP \mid right \rightarrow Right \rightarrow up \rightarrow STOP)$ □

X2 一台机器同时可提供两种方法破开五便士的硬币(比较 1.1.2 中 X3 和 X4，在这两例中没有选择余地)，此机器的行为可定义为

$CH5C = in5p \rightarrow (outlp \rightarrow outlp \rightarrow outlp \rightarrow out2p \rightarrow CH5C$

$\quad \mid out2p \rightarrow outlp \rightarrow out2 \rightarrow CH5C$

由换钱的顾客进行选择。 □

X3 一台售货机，其每笔交易或卖巧克力或卖太妃糖，定义为

$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$ □

X4 一台更为复杂的售货机，在可使用的硬币、出售的货品和找钱的方式上，都有选择余地，可定义为

$VMC = (in2p \rightarrow (large \rightarrow VMC \mid small \rightarrow outlp \rightarrow VMC)$

$\quad \mid inlp \rightarrow (small \rightarrow VMC \mid inlp \rightarrow (large \rightarrow VMC$

$\quad \mid inlp \rightarrow STOP)))$

象许多复杂的机器一样，这台售货机有一个设计上的缺陷，顾客不能接连投入三个便

[33] 字母表都是一致的假设很重要。因为后续的 P 或者 Q 进程里包括一个递归定义，因此 P 的字母表里必须有 x 事件。

士的硬币[34]。但是改一下用户手册要比改进机器容易的多，所以，我们在机器上加上一条注意事项

"注意：不要接连投入三个便士。" □

X5 一台售货机允许顾客先品尝一块巧克力，然后再付钱。它也允许先付后尝。此进程写为

VMCRED = μX • (coin → choc → X | choc → coin → X) □

X6 为了避免损失，使用 VMCRED 时必须先付使用费。改良的进程如下

VMS2 = (coin →VMCRED)

这台售货机最多可允许连续两次投入硬币，然后最多连续送出两块巧克力；但是先付了多少钱，售货机不会送出更多的巧克力，决不多给。 □

X7 一个拷贝进程执行下列事件：

in.0 由进程的输入通道输入 0

in.1 由进程的输入通道输入 1

out.0 由进程的输出通道输出 0

out.1 由进程的输出通道输出 1

这一进程的行为是成对事件的重复发生。在每个周期里，它输入一位数，又将这一位数输出，记为

COPYBIT = μX • (in.0 → out.0 → X | in.1 → out.l → X)

注意这一进程允许由环境选择输入的值，但输出时环境就不能选择(影响)了。在第四章中我们定义和处理通信问题时，这一点是输入和输出的主要区别。 □

选择的定义可以很容易的推广到两个"选择"以上的情形，即

(x → P | y → Q | … | z → R)

注意选择符号 | 不是进程之间的算子；因此对进程 P 和 Q，P | Q 的写法是有语法错误的[35]。不能这样写的原因是，我们想回避给表达

(x → P) | (x → Q)

赋予任何含义，这个表达式看上去好象向环境提供了选择第一个事件的权利，但实际

[34] 观察 VMC 的行为定义，如果依次投入 3 个便士，进程的行为序列会走到 STOP 的地方。机器不再工作。

[35] CSP 的 Choice 需要显示的用(x → P | y → Q)的方式来表达对不同的事件(x 或者 y)，相应的不同的处理进程。

上又没有做到这一点[36]。解决这个问题的方法是引入非确定性，这将在 3.3 节中介绍。

现在，假设 x, y, z 是不同事件，则

$$(x \to P \mid y \to Q \mid z \to R)$$

应被看作是单个算子，这个算子有三个变量 P,Q,R。千万不能把它

$$(x \to P \mid (y \to Q \mid z \to R))$$

混为一谈，后者犯有语法错误[37]。

总之，设 B 为事件集合，对 B 中每个 x，表达式 P(x)定义了一个进程，则

$$(x{:}B \to P(x))$$

定义了这样一个进程，开始时，它允许选择集合 B 中任意事件 y，然后按 P(y)继续执行。这个进程定义表达式应读作“从 B 中选择 x 然后按进程 P(x)动作”。表达式中 x 是局部变量，因此有

$$(x{:}B \to P(x)) = (y{:}B \to P(y)$$

集合 B 定义了进程的初始菜单[38]，因为它给出了进程启动时，可以选择的动作的集合。

例子

X8 一个进程，它在任何时候都能执行其字母表 A 中的任何事件，则它为

$$\alpha RUN_A = A$$

$$RUN_A = (x{:}A \to RUN_A)$$ □

如果在初始菜单内只含一个事件 e 时，则由于 e 是唯一可能的初始事件，就有

$$(x{:}\{e\} \to P(x)) = (e \to P(e))$$ [39]

在一种更为特殊的情形里，初始菜单是空的。这时，不会发生任何事件，因此有

$$(x{:}\{\} \to P(x)) = (y{:}\{\} \to Q(y)) = STOP$$[40]

二元选择算子|也可以用下面更为一般的记法来定义

$$(a \to P \mid b \to Q) = (x{:}B \to R(x))$$

---

[36] 假设环境触发了事件 x，系统不知道该执行 P 还是 Q。引起混乱。

[37] (y → Q | z→ R)是一个进程。不能直接跟在 Choice"|"后面。属于语法错误。在"|"后面，需要首先是一个 event。

[38] 类似面向对象语言里，一个对象对外的 Public 的 Method 方法。通过这些方法，一个对象可以被触发，或者接受外部的调用。

[39] 当选择事件是唯一时，选择就变成了前缀。换言之，前缀是选择的特殊形式。

[40] STOP 进程的一个形式定义。不接受任何外部的触发，不响应任何外部的事件了。

这里 B = {a, b}, 而且 R(x) = if x = a then P else Q

在三个或三个以上对象之间的选择可以类似地表述出来。这样，我们把选择、前缀和 STOP 定义为一般选择记法的特殊情形。这在我们制定进程必须遵从的一般法则(1.3 节)，以及进程实现(1.4 节)中将大有用处。

1.1.4 联立递归(Mutual Recursion)

通过递归手段可以把单个进程定义为一个单一方程和此方程的解[41]。我们可容易地把它推广到：多个未知的进程是联立方程组的解。这个推论要成立的条件是：联立方程的右侧都必须为卫式的，且每个未知进程都在某一个方程的左侧恰好出现一次。

例子

X1 饮料机上有两个分别标为橘子汁(ORANGE)和柠檬汁(LEMON)的按键。两个按键触发的动作定义为 setorange 和 setlemon。给出饮料的动作定义为 orange 和 lemon。饮料机最后吐出那种饮料决定于客户按下了那个按键。以下是定义三个进程的字母表和行为的方程

$\alpha DD = \alpha O = \alpha L = \{setorange, setlemon, orange, lemon\}$

$DD = (setorange \rightarrow O \mid setlemon \rightarrow L)$

$O = (orange \rightarrow O \mid setlemon \rightarrow L \mid setorange \rightarrow O)$

$L = (lemon \rightarrow L \mid setorange \rightarrow O \mid setlemon \rightarrow L)$

非形式地说，当第一个事件发生后，饮料机就处于 O 状态或 L 状态。在这两种状态下，它或者给出相应的饮料，或者还可以改变想法转换到另一选项去。重复选择是可以的，但没有任何印象，因为是一个简单的递归重复。 □

采用带下标的变量，我们就可以定义无穷方程组。

X2 一个对象以地面为起点，可以向上运动。以后任意时刻它都可以向上或向下运动，除非它落到地上，不能再向下落了。而且规定当它在地上时，可向四周运动。假设自然数 $n \in \{0, 1, 2...\}$。对每个 n，我们引入一个加下标的名字 CT_n，用它来表述这个对象离地第 n 次运动的行为方程。对象的初始行为定义为

$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow cT_0)$

[41] $\mu X : A \bullet F(X)$

剩下的无穷方程组为

$$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow cT_n)$$

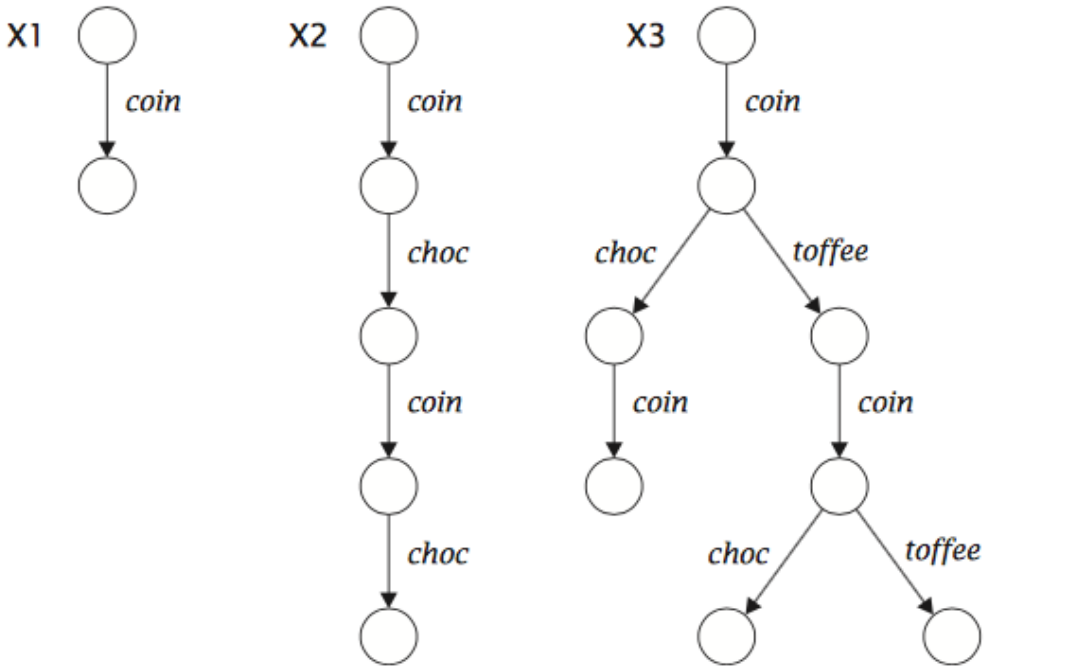
其中 n 为自然数 0, 1, 2...

归纳定义的方程式是否有效，一般情形下需要看每个方程右侧所用下标是否比左侧用的小。而这里，CT_{n+1} 却是由 CT_{n+2} 给出的。因此，这个定义式只能被看做是一个无穷的联立递归定义方程组，它是否有效就要看每个方程右侧是否是卫式。 □

1.2 示意图(Pictures)

用树形结构图作为进程行为的示意图，有时是很有帮助的，树形图由箭头连起来的一些小圆圈组成，在状态机的传统术语里，这些圆圈代表进程的各个状态，箭头表示状态间的转换。树根上的圆圈(一般画在树形图上方)是初始状态；整个进程沿着箭头方向向下进行。每个箭头旁边标有实现状态转换时发生的事件。在由同一节点引出的不同的箭头或者有向边必须是不同的事件触发的。

例子 (1.1.1 节中 X1, X2; 1.1.3 节中 X3)



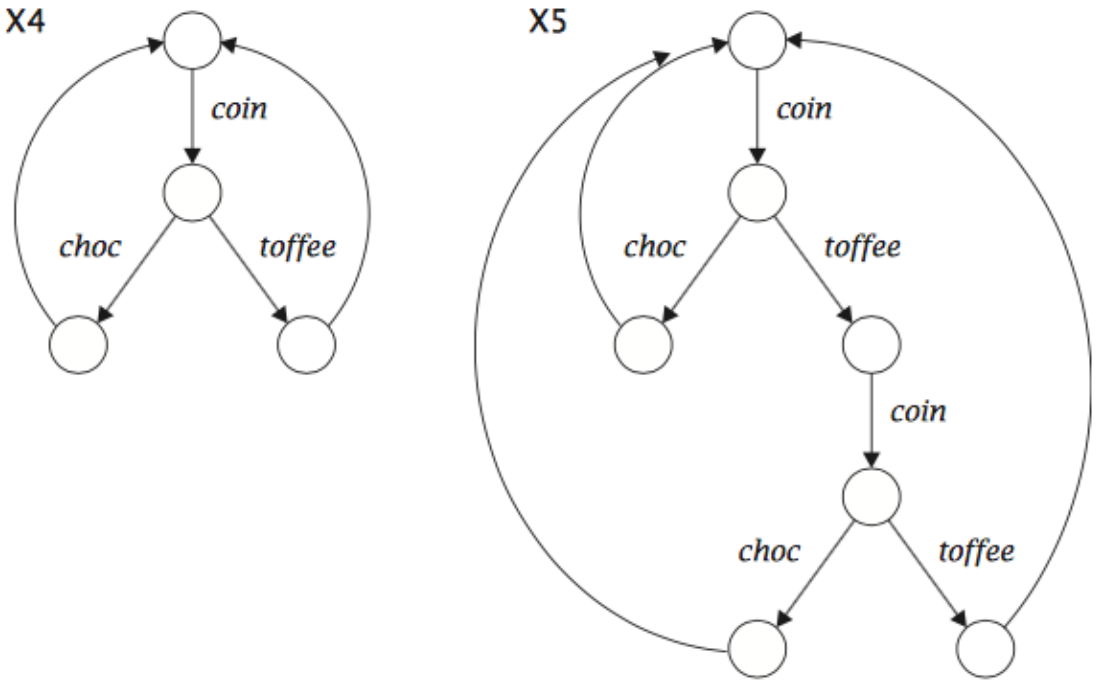
□

在这三个例子中，每个树的各个分支都以 STOP 结束，STOP 表示为一个不引出箭头的圆圈。如果用图示法表示具有无限行为的进程，我们得引入另一个约定，即有

一种没有任何标记的箭头，它可沿树上叶点绕回到前面的某个圆圈去。这个约定是说，当一个进程走到了在这种箭头的尾部所在的节点时，它就立即神不知鬼不觉地回到该箭头所指向的节点[42]。

显然，下面两个不同的示意图所描绘的是同一个进程(1.1.3 节的 X3)。然而，要用图示的方法来证明它们是相同的[43]，则是很不容易的事。这是图示法的一个弱点。

图示法的另一弱点是不能描绘具有很多很多个或有无穷个状态的进程，例如，我们做联立方程组里的例子 CT_0。

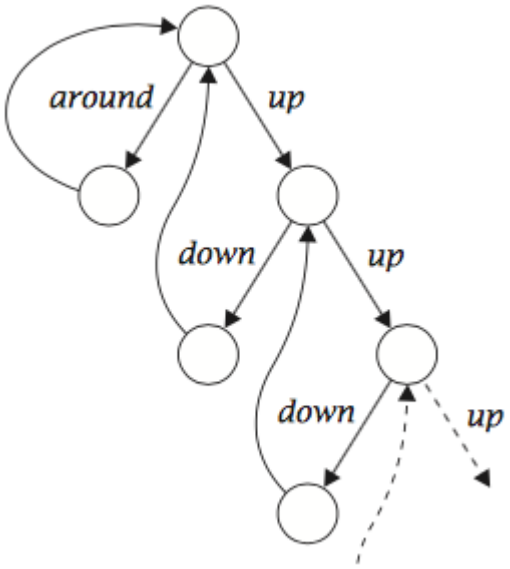


□

要画出 CTo 整个示意图，实在没有那么多地方。只有 65536 种不同状态的筹码进程的示意图，就够我们画好一阵了。

[42] 如果指向根节点，就是一个递归进程的行为。例如，前面谈及的自动售货机。

[43] X4 与 X5 都是对 1.1.3 节的 X3 的进程 VMCT 进程行为的图示化。X5 与 X4 的区别是：X5 做了一次递归迭代展开，然后才循环回到根节点的。这种人的理性可以很简单判断出来的两个图是等价的关系，对于图计算来说可能可以很复杂。



1.3 法则(Laws)

即使目前为止我们介绍和使用了相对严格的符号系统，我们发现要刻画一个进程的行为，还是存在许多不同的方法。譬如，被选择的事件的表述次序，显然不应该有多大关系，即

$$(x \rightarrow P \mid y \rightarrow Q) = (y \rightarrow Q \mid x \rightarrow P)$$

而另一方面，能动作的进程和什么都不能做的进程显然不等同。

$$(x \rightarrow P) \neq STOP$$

要正确理解符号的含意，且能准确无误地使用它们，就必须学会识别哪些表达式刻画了同一对象，哪些没有。这就象懂代数的人都知道(x+y)和(y+x)代表同一个数。我们可以运用一些代数法则证明或者证伪两个字母表相同的进程的等同性。这些关于进程的代数法则与算术中的法则很类似。

第一条法则(L1)是处理选择算子的(1.1.3 节)。它说明两个由选择算子定义的进程，如果它们第一步的选择就不同，或者第一步是相同的，但以后的行为不同，则这两个进程就不同。但是如果两个进程的初始选择相同，其后续所以的行为也相同，则这两个进程显然是一样的[44]。

L1 $(x : a \rightarrow P(x)) = (y : b \rightarrow Q(y))$

$$\equiv (A = B \wedge \forall x \in A \bullet P(x) = Q(x))$$

在本书的任何地方，我们都假设方程式两边的进程的字母表都相同，这一点以后不再单独说明。

[44] 从状态图的角度，就是两个进程的状态机对各个事件的响应是完全一致的。两个图是等价的。

法则 L1 有几个推论：

L1A STOP ≠ (d → P)

证明 LHS = (x:{} → P)　　由定义(1.1.3 节结尾)[45]

≠(x:{d} → P)　　∵ {} ≠ {d}

=RHS　　由定义(1.1.3 节结尾)

L1B (c → P) ≠ (d≠Q)　　如果 c≠d

证明 {c} ≠ {d}[46]

L1C (c → P | d → Q) = (d → Q | c → P)

证明

定义

R(x) = P　　当 x=c 时

= Q　　当 x=d 时

LHS = (x:{c, d} → R(x))　　由定义[47]

= (x:{d, c} → R(x))　　∵ {c, d} = {d, c}

= RHS　　由定义

L1D (c → P) = (c → Q) ≡ P = Q

证明 {c} = {c}[48]

用以上法则可证明一些简单的命题。

例子

X1 (coin→ choc → coin→ choc → STOP) ≠ (coin→ STOP)

证明 由 L1D，再由 L1A 即可得证。 □

X2 μX • (coin → (choc → X | toffee → X))

= μX • (coin → (toffee→ X|choc → X))

证明由 L1C 即可得证。

为了证明递归定义进程方面更普通的命题，我们还需引入一条新法则，即每个卫式递归方程有唯一解。

---

[45] STOP 进程定义为不响应任何事件的异常进程。不会有任何事件能够触发进程，并进行到 P 的过程。

[46] 根据 L1，最开始的事件不一样，整个进程的行为当然不一样。

[47] 参阅 1.1.3 节关于 Choice 的讨论。

[48] 根据 L1 的法则，如果起初事件集合相同，并且整个进程的行为也相同，那么初始事件之后的进程行为必须严格相同。因此，对于 L1D 而言， 起初事件只有相同的 c，显然 P 必须等价于 Q 如果(c → P) = (c → Q)。反之亦然。

L2 如果 F(X)是卫式表达式，则有

$(Y = F(Y)) \equiv (Y = \mu X \bullet F(X))$

紧接着一个重要推论是，$\mu X \bullet F(X)$确实是有关方程的解：

L2A $\mu X \bullet F(X) = F(\mu X \bullet F(X))$[49]

X3 假设 VM1 = (coin → VM2), 且 VM2 = (choc → VM1), 要证明 VM1=VMS[50]。

证明

VM1 = (coin → VM2)　　根据 VM1 定义

= (coin → (choc → VM1))　　根据 VM2 定义

所以，VM1 是与 VMS 递归方程完全一样的进程方程的一个解。由于方程式是卫式的，因此它只有一个唯一解。于是，VM1 和 VMS 只是该唯一解的两个不同名字而已。

这个命题的成立很显然的，不需要通过证明来增加可信度。该证明的唯一目的是要通过例子，来说明这些法则的强大，可以用来证明这类事实。当我们证明某些显而易见的事实，而使用的法则却没有那么明显时，要详细检查每一步证明，防止出现循环论证。

法则 L2 可推广到联立递归。我们可以通过使用下标的方式把联立递归方程组的一般形式记为

$X_i = F(i, X)$　　对所有 $i \in S$

这里

S 是下标集合，每个方程的下标是 S 的一个元素；

X 是进程数组，其下标变化范围为 S；

F(i, X)是卫式表达式。

在这些条件下，法则 L3 说明只存在一个唯一的数组 X，其元素满足所有方程。即

L3 在上述条件下，

if $(\forall\ i : S \bullet (Xi = F(i, X) \wedge Y_i = F(i,Y)))$ 则 $X = Y$[51]

---

[49] Y = F(Y))方程有唯一解，这个解是 Y = μX • F(X)。把解代入方程，即得到 μX • F(X) = F(μX • F(X))

[50] VMS 方程定义和递归形式定义可参阅 1.1.2 节的 X2。VMS =(coin → (choc → VMS))

1.4 进程的实现(Implementation of processes)

到目前为止，可表述的进程可表示为以下形式

(x:b →F(x))

其中 F 是符号到进程的对应函数，集合 B 可以是空集(如 STOP), 或只有一个元素(如前缀情形)，或多个元素(例如，多种选择的情形)[52]。对递归定义的进程，我们一直强调所用递归式必须是卫式的。因此，递归定义的进程可形式化的记为

μX • (x:b → F(x,X))[53]

而且，应用 L2A，这个表达式可按要求的形式展开

(x:b → F(x, μX • (x:b → F(x,X)))[54]

这样每个进程均可被看作是一个在函数域 B 上的函数 F。其中 B 定义了进程一开始准备执行的事件集合；对 B 中的每个 x，F(x)定义了进程执行第一个事件之后的未来行为。

根据上述观点，每个 CSP 的进程可以通过适当的函数式程序设计语言中的函数来表达，譬如 LISP 语言。进程字母表中的每个事件可表述为一个 LISP 里的原子，譬如"COIN，"TOFFEE。一个进程是一个可以把这些院长事件作为输入参数的函数。如果这个符号不是该进程的第一个(批)可能执行的事件，这个函数则给出一个特殊符号“BLEEP”作为结果。符号“BLEEP”的用途仅在于此。例如，既然 STOP 从不执行任何事件，“BLEEP”就是它能给出的唯一的结果，因此它被定义为

STOP = λx • "BLEEP

如果自变量的值是该进程的可能事件，这个 LISP 函数则给出另一函数作为结果，代表进程的后续行为。于是，(coin → STOP)就可用如下 LISP 函数表达

---

[51] 可以简单证明如下：通过联立递归方程的定义，每个 X_i 有唯一解，并为 μX • F(i, X)；同理，每个 Y_i 有唯一解，并为 μY • F(i, Y)。因为两个方程组一模一样，而且方程的解是唯一的，因此， X = Y。

[52] 本节关于 Lisp 实现的讨论就是按照 STOP 进程，前缀，选择和递归进程来逐步介绍的。

[53] 因为对于一个递归进程，非方程的 μ 的形式化描述是 μX • F(X)。在这里，我们把 F 可以表达为一种通用多项选择(x:b → F(x))的符号记法代入，于是得到， μX • (x:b → F(x,X))。要注意的是大写的 X 是代表递归进程的符号(名字)；小写的 x 是代表了进程响应的各种事件。

[54] 对于是卫式递归的通用进程的方程式，方程式右侧显然(x:b → F(x, X))。根据 L2A， 其意思就是把递归方程的唯一解 μX • (x:b → F(x,X))，可以代入到方程式里替换所有出现该递归进程名(符号)的地方，于是方程式的右边得到：(x:b → F(x, μX • (x:b →F(x,X)))。例如，对于一个自动售货机 VMCT(1.1.3 X3)，X=μX • (coin → (toffee→ X|choc → X))，我们可以理解(coin → (toffee → X|choc → X))为这里的 F(x, X)。显然，进程可以简单的迭代展开一次变成：(coin → (toffee → ((coin→ (toffee → X|choc → X))|choc → ((coin → (toffee → X|choc → X))))))。理论上，可以无限制的展开。

λx • if x = "COIN then

STOP

else

"BLEEP

这个例子利用了 LISP 的一个便利条件，即可以返回一个函数(如 STOP)作为函数结果。LISP 的另一个便利之处是，它允许将一个函数做为一个自变量传递给另一个函数，我们利用这一点来表达通用的前缀操作(c → P)

prefix(c, P) = λx • if x =c then

P

else "BLEEP

一个 LISP 函数如果要表示一个一般的二元选择算子(c → P | d→ Q)，需要四个参数：

choice2(c, P, d, Q) = λx • if x= c then

P

else if x= d then

Q

else

"BLEEP

我们可以利用 Lisp 的 LABEL 特性表达 CSP 的递归进程。例如，简单自动售货机

(μX • coin → choc → X)可以表示为

LABEL X • prefix("COIN, prefix("CHOC, X))

LABEL 也可以用来表示联立递归式。如 CT(1.1.4 节 X2)就可被看作是由自然数到进程的一个函数，当然进程本身也是函数，但先不考虑这一点。这样，CT 就可以定义为

CT = LABEL X • (λn • if n=0 then

choice2("AROUND,X(O),"UP,X(1))

else

choice2("UP, X(n+1), "DOWN, X(n-1))

CT(0)是以地面为起点的进程。整个进程组从 CT(0)开始。

如果 P 是表示一个进程的函数，且 A 为包含该进程字母表中符号的表，则 LISP 函数 menu(A,P)定义为给出可做为 P 的第一个事件的全部符号。

menu(A, P) = if A=NIL then

```
                NIL
                else if P(car(A)) = "BLEEP then
                                menu (crd(A)
                        else
                                cons(car(A), menu(cdr(A), P))
```

假设 x 在 menu(A, P)中，且 P(x)不是"BLEEP，那么 P(x)是一个函数，它定义了 P 执行完 x 事件后的未来行为。由此推出，如果 y 在 menu(A, P(x))中，则 P(x)(y)定义的是在 x 和 y 都发生之后，进程的后续行为。这条规律为我们探讨进程的行为提示了一个很有用的方法。编写一段程序，先将 menu(A,P)的值输出到屏幕上，然后再由键盘输入一个符号，观察结果。如果这个输入的符号不在表 menu(A,P)中，则就会发出清晰的响声，但不接受它。如果它在这个表中，就会被接受，然后用键盘输入的符号产生的结果，例如，P(x)来取代 P，并一直重复下去。当结束这一过程时键入符号"END。于是，如果 k 是由键盘输入的符号序列，则以下函数给出相关的输出序列

```
interact(A, P, k) =
        cons(menu(A, P), if car(k) = "END then
                NIL
           else if P(car(k))= "BLEEP then
                cons( "BLEEP, interact (A, P, cdr(k) ) )
                else
                        interact (A, P(car(k), cdr(k)))
```

以上定义 LISP 函数时所用的记号法很不正规，还需要把它们翻译成具体的 LISP 的 S-表达式的形式。例如在 LISPkit 中，前缀函数可定义为

```
(prefix
        Lambda
        (a  p)
        (lambda (x)(if (eq x a) p (quote BLEEP))))
```

幸运的是，我们使用的只是纯函数式 LISP 语言的一个很小的子集，因此，在各种类型机器上用各种 LISP 变种翻译并运行进程时，都不会有什么困难。

如果有好几种 LISP 版本都能用，我们应该选择带有变量静态约束的那种 LISP，使用它时会方便些。而使用惰式求值的 LISP 还要更方便些，因为这种 LISP 允许对递

归方程直接编码，不需用 LABEL 特性，因此有

VMS = prefix("COIN, prefix("CHOC, VMS))

如果输入和输出是由惰式求值的 LISP 实现的，则在调用函数 interact 时以键盘作为它的第三个参数；进程 P 的菜单则是它的第一个输出。使用者通过在不断输出菜单中选择并输入符号，就可以交互式地研究进程 P 的行为。

在其它几种 LISP 版本中，函数 interact 应该改写，使用显示输入和输出来达到同样的效果。总之，这样一来，我们就有可能观察计算机执行一 LISP 函数表示的任一进程了。从这个意义上来说，这种 LISP 函数就可以被看做是其相应进程的实现。更进一步地说，象 prefix 这样的 LISP 函数，它们作用于表示进程的函数上，就可以被看作是相应的进程算子的一种实现。

1.5 迹(Traces)

一个进程行为的迹是进程事件被触发执行的一个有限序列，这个符号序列记录该进程到某一时刻为止执行的各个事件。设想有位观察员带着笔记本，观察这个进程，每一个事件发生时他就将该事件名记下来。我们完全可以忽略两个事件同时发生的可能性；因为即使真有两个事件同时发生了，观察员做记录时也总要有先有后，而且他记录的先后次序也不是重要的。

一个迹表示为事件符号的一个序列，中间由逗号断开，并且括在一对角括号内。

<x, y>由两个事件组成，y 跟在 x 后发生。

<x>只包含一个事件 x 的序列。

< >不包含任何事件的空序列。

例子

X1 简单自动售货机 VMS(1.1.2 节中 X2)若刚好完成了为两位顾客服务，这时它的迹为

(coin, choc, coin, choc) □

X2 同一个售货机，在第二位顾客的巧克力被送出之前的迹为

(coin, choc, coin)

不论是进程还是观察员都不具备一个完整的交易这种概念。顾客的饥不可耐和机器满足顾客要求的责任感都不在进程的字母表内，所以也就不能被观察到或记录下来。□

X3 在进程开始执行任何事件之前，观察员的笔记本是空白的。这种情况表示为空迹

< >

空迹是每个进程可能的最短的迹。 □

X4 复杂自动售货机 VMC(1.1.3 节中 X4)有以下七个长度不超过 2 的迹

< >

< in2p> <inlp>

<in2p, large> <in2p, small> <inlp, inlp> <inlp, small>

对一台给定的机器来说，那四个长度为 2 的迹中只有一个能真正发生。至于选择哪一个，就要有第一位使用这台售货机的顾客来决定了[55]。 □

X5 如果第一位顾客不理睬 VMC 机器的注意事项[56]，那么 VMC 的迹可以是

<inlp, inlp, inlp >

这个迹并不记录机器的损坏。机器的损坏是通过下述事实表现出来的，即在这台机器所有可能的迹中，不存在一个迹是这个迹的延伸，也就是说，不会存在事件 x，能使

<inlp, inlp, inlp, x>

成为 VMC 可能的迹。这时，顾客可能干着急没办法，观察员也是一筹莫展，不会再有事件发生了，不会再有符号能记在小本上了。至于顾客和售货机的结局如何，我们就不得而知了，因为这不在我们选择的字母表内。

1.6 迹的运算(Operations on traces)

在对进程行为进行记录，描述以及正确理解过程中，迹扮演着一个很关键的角色。在这一节里，我们要探讨迹的一般性质以及迹的运算。我们要用到以下约定

s, t, u 表示迹

S, T, U 表示迹的集合

f, g h 表示函数

1.6.1 连接(Catenation)

迄今最重要的迹运算是迹与迹之间的连接，即把一对操作数 s 和 t 按照 s, t 的次序简单的拼到一起，构成一个迹，记为 s ^

[55] VMC 的递归行为定义是：VMC = (in2p → (large → VMC | small → outlp → VMC) | inlp → (small →VMC | inlp → (large → VMC | inlp → STOP)))。支持多种选择。但一旦选择了，执行轨迹就不一样了。

[56] VMC 机器的注意事项：“不要接连投入三个便士。”

例如

<coin, choc >^<coin, toffee> = <coin, choc, coin, toffee>

<inlp> ^<inlp> = <inlp, inlp>

<inlp, inlp> ^ < > = <inlp, inlp>

连接的最重要的性质是，连接是满足结合律的(associative)，且其单位元素为< >。

L1 s^ < > = < > ^ s = s

L2 s ^ (t ^ u) = (s ^ t) ^ u

以下法则是很显然的，而且非常有用

L3 s ^ t = s ^ u ≡ t = u

L4 s ^ t = u ^ t ≡ s = u

L5 s ^ t = < > ≡ s = < > ∧ t = < >

假设 f 是一个映射迹到迹之间的函数。如果它对空迹的映射结果仍然为空迹，即 f(< >) = < >，则函数 f 被称作是严格的(strict)。如果 f(s ^ t)=f(s) ^ f(t)，则说函数 f 是满足分配律的(distributive)。所有可分配的函数都是严格的[57]。

设 n 为自然数，我们定义 t^n 为 t 的 n 次自相连接。对 n 进行归纳，即有结果

L6 t^0 = < >

L7 t^{n+1} = t ^ t^n

上述两项很有用的法则，实际上就是定义本身；以下是另外两个可由它们推导证明的法则

L8 t^{n+1}=t^n ^ t

L9 $(s \hat{} t)^{n+1}$ = s ^ $(t \hat{} s)^n$ ^ t

1.6.2 局限(Restriction)

表达式(t ↾ A)表示迹 t 局限于集合 A 中的符号，即把 t 中所有不属于 A 的符号去掉后留下的迹。例如

<around, up, down, around> ↾ {up, down} = <up, down>

局限运算是可分配的，因此也是严格的。相关一些法则

L1 <> ↾ A = <>

L2 (s ^ t) ↾ A = (s ↾ A) ^ (t ↾ A)

[57] f(<>) = f(<> ^ <>)。因为 f 满足分配律，所以，f(<>) = f(<>) ^ f(<>)。因此，f(<>) = <>。因此是严格的。

对于单元素(Singleton)序列，显然有

L3 $\langle x\rangle \upharpoonright A = \langle x\rangle$　如果 $x \in A$

L4 $\langle y\rangle \upharpoonright A = \langle\rangle$　如果 $y \notin A$

一个满足分配律的函数，在定义了它对单元素序列的作用效果后，这个函数整个的行为也就唯一地确定了。因为当作用于更长的序列时，我们总可以使它分别作用于序列的每个元素。然后把各项结果连接起来，然后可得到我们所要的结果。例如，如果 $x \neq y$。

$\langle x, y, x\rangle \upharpoonright \{ x \}$

$= (\langle x\rangle \hat{} \langle y\rangle \hat{} \langle x\rangle) \upharpoonright \{ x \}$

$= (\langle x\rangle \upharpoonright \{ x \}) \hat{} (\langle y\rangle \upharpoonright \{ x \}) \hat{} (\langle x\rangle \upharpoonright \{ x \})$　[利用法则 L2][58]

$= \langle x\rangle \hat{} \langle\rangle \hat{} \langle x\rangle$　[利用法则 L3 和 L4]

$= \langle x, x\rangle$

以下是关于局限与集合运算间的关系的法则。一个迹局限于空符号集的结果是一个空迹，连续受限于两个集合的迹的结果就跟这个迹受限于这两个集合的交集的结果相同。对迹 s 的长度进行归纳，就能严格证明这几项法则

L5 $s \upharpoonright \{\} = \langle\rangle$

L6 $(s \upharpoonright A) \upharpoonright B = s \upharpoonright (A \cap B)$

1.6.3 首部与尾部(Head and tail)

假设 s 为一非空序列，取它的第一个符号为 s_0，去掉 s_0 后结果记为 s′。如

$$\langle x, y, x\rangle_0 = x$$

$$\langle x, y, x\rangle = \langle y, x\rangle$$

这两个运算对空列无定义。

L1 $(\langle x\rangle s)_0 = x$

L2 $(\langle x\rangle s)' = s$

L3 $s = (\langle s_0\rangle \hat{} s')$　如果 $s \neq \langle\rangle$

法则 L4 给出证明两个迹相等的简便的方法。

L4 $s = t \equiv (s = t = \langle\rangle \vee (s_0 = t_0 \wedge s' = t'))$[59]

---

[58] 通过分配律作用在每个元素上。

[59] 证明很简单，如果 s 和 t 都是空串，显然 s = t；或者，如果 s 的第一个符号和后续的符号都与 t 的第一个符号和后续的符号相同，显然，根据 L3，s = t。

1.6.4 星号(Star)

我们定义集合 A^*是由符号表 A 符号构成的所有的有穷迹(包括< >)的集合。因此，当局限于字母表 A 时，这些迹都保持不变。通过这个性质我们可引出一个简单定义

$$A^* = \{s \mid s \restriction A = s\}$$

以下法则是这个定义的推论。

L1 $< > \in A^*$

L2 $< x > \in A^* \equiv x \in A$

L3 $(s \frown t) \in A^* \equiv s \in A^* \wedge t \in A^*$

这些法则很有力，足以确定一个迹是否是 A^*的一个元素。例如，如果 $x \in A$ 且 $y \notin A$，则有

$< x, y > \in A^* \equiv (< x > \frown < y >) \in A^*$

$\equiv (< x > \in A^*) \wedge (< y > \in A^*)$　　由 L3

$\equiv$ true $\wedge$ false　　由 L2

下一法则可作为 A^*的一个递归定义。

L4 $A^* = \{t \mid t = < > \vee (t_0 \in A \wedge t' \in A^*)\}$

1.6.5 次序(Ordering)

假设 s 是序列 t 的一个初始子序列，则我们总能找到 s 的某个扩展序列，使 $s \frown u = t$。因此，我们定义一种次序关系

$$s \leq t = (\exists\ u \bullet s \frown u = t)$$

我们定义这种关系为 s 是 t 的一个前缀。例如

$< x, y> \leq <x, y, x, w>$

$< x, y> \leq <z, y, x> \equiv x = z$

如法则 L1 至 L4 所述，关系≤是个偏序关系[60]，其最小元为< >。

L1 $< > \leq s$　　最小元

L2 $s \leq s$　　自反性

L3 $s \leq t \wedge t \leq s \Rightarrow (s = t)$　　反对称性

[60] 偏序关系的含义是，给定字母表 A 组成的任意两个序列，不是任何两个序列都必须需要保证存在≤这种逻辑关系。可以是谁也不是谁的前缀。例如，〈x y z　m n〉与〈z x n m〉，不存在这种前缀关系。

L4 $(s \le t \wedge t \le u) \Rightarrow (s \le u)$ 传递性

下一法则与 L1 并用，给出计算 $s \le t$ 是否成立的方法。

L5 $(< x > ^\frown s) \le t \equiv t \neq < > \wedge x = t_0 \wedge s \le t'$[61]

对一给定序列，它的所有的前缀是全序的(Total Ordering)[62]。

L6 $s \le u \wedge t \le u \Rightarrow s \le t \vee t \le s$[63]

假设 s 为 t 的一个子序列(不一定是初始子序列)，我们就说 s in t；记为

L7 $s \text{ in } t = (\exists\ u,v \bullet t = (u ^\frown s ^\frown v))$

这个关系也是个偏序关系，满足上述 L1 至 L4[64]。它还满足下述法则

L8 $(< x > ^\frown s) \text{ in } t \equiv t \neq < > \wedge ((t_0 = x \wedge s \le t') \vee (< x > ^\frown s) \text{ in } t')$

我们定义一个函数单调的性质。定义为如果迹到迹的函数 f 能够保持前缀的次序关系 $\le$，也就是说

$f(s) \le f(t)$　　如果有 $s \le t$[65]

所有满足分配律的函数都是单调函数。例如

L9 $s \le t \Rightarrow (s \restriction A) \le (t \restriction A)$

一个二元函数对它的两个自变量中的一个可能是单调的(这时令另一个自变量不变)。例如，连接对第二个自变量(但不能是第一个)是单调的。

L10 $t \le u \Rightarrow (s ^\frown t) \le (s ^\frown u)$

对所有自变量均为单调的函数称为单调函数。

1.6.6 长度(Length)

迹 t 的长度记做#t。例如#< x, y, x > =

定义长度运算#的法则如下

L1 #< > = 0

L2 #<x> = 1

L3 $\#(s ^\frown t) = (\#s) + (\#t)$

A 中符号在 t 中出现的个数为$\#(t \restriction A)$

---

[61] 只需确保<x>和 s 的连接形成的序列中，x 是 t 的第一个字符，s 是 t 除了第一个字符之后的子串。

[62] 任何两个前缀，彼此之间一定存在前缀关系。不存在孤立的两个前缀，无法定义其关系。

[63] 要么 s 是比 t 还靠前的前缀，或者，t 更靠前。

[64] L1 〈 〉 in s 最小元 L2 s in s 自反性 L3 s in t ∧ t in s → (s = t)反对称性 L4 (s in t ∧ t in u) → (s in u)传递性

[65] 可以这样理解，通过函数 f 的变换，前缀的次序关系不会丢掉。

L4 #(t ⌈ (A ∪ B)) = #(t⌈(A)+#(t⌈B) - #(t⌈(A∩B))

L5 s ≤ t ⇒ #s ≤ #t

L6 #(t^n) = n × (#t)

符号 x 在迹中 s 中出现的次数定义为 s↓X = #(s ⌈(x})

1.7 迹的实现(Implementation of traces)

为了在计算机内表示迹并实现对它的运算，我们需要一种高级的表处理语言。幸运的是，LISP 能满足我们的需要。可以用代表事件的原子组成的 Lisp 表，能很容易地把迹表示出来。

< > = NIL

<coin> = (cons("COIN, NIL))

<coin, choc> = "(COIN CHOC)

其含义为 cons("COIN, cons("CHOC, NIL))

迹的运算可轻易的通过 Lisp 表的函数的功能来实现。例如，一个非空表的首部和尾部可以由原函数 car 和 cdr 给出

t_0 = car(t)

t' = cdr(t)

< x> ^ s = cons(x, s)

我们用大家熟悉的 append 函数来实现一般的连接运算。append 函数是由下述递归式给出定义的

s ^ t = append(s, t)

其中，

append(s, t) =

if s = NIL then t

else

cons(car(s), append(cdr(s), t))

这个定义的正确性由下述法则为依据

< > ^ t = t

s ^ t = <s_0> ^ <s ^ t>其中 s ≠ < >

LISP 的 append 函数的终止可以通过这样的事实保证，即每次递归调用时用作第一个自

变量的表都要比它前一级递归调用时的短[66]。自变量的类似特性也保证以下定义的其它运算实现的正确性。

为实现局限(Restricted)运算，我们用有限集合 B 的元素的表代表 B 本身，检验(x ∈ B)可通过调用函数来实现

```
ismember(x, B) =
        if B = NIL then
        false
        else if x = car (B) then
        true
        else ismember(x, cdr(B))
```

(s↾B)可由下列函数实现

```
restrict(s,B) =
    if s= NIL then
    NIL
    else if ismember(car(s), B) then
        cons (car(s), restrict(cdr(s), B) )
        else
            restrict (cdr(s), B)
```

检验(s≤t)可由给出结果是 true 或 false 的函数来实现；根据 1.6.5 节的 L1 和 L5 这个函数可定义为

```
ismember(x, B) = if s = NIL then
        true
        else if t = NIL then
        false
        else
            car(s) = car (t) and
        isprefix(cdr(s)，cdr(t))
```

1.8 进程的迹(Trace of a process)

在 1.6 节中，我们定义进程的迹是进程到某一个时刻为止的行为的顺序记录。进程开始前，谁也不知道进程的哪个可能的迹将会被记录下来：这个选择是由不受进程

[66] 最后会遇到递归方程的控制条件：if s = NIL then t 。

约束的外部环境因素来决定的。然而，进程 P 的所有可能的迹的全集是能事先知道的，我们定义函数 traces(P)，用这个函数来表示产生一个进程可能的行为(迹的集合。例子

X1 进程 STOP 的行为只有一个迹，即< >。记录这个进程的观察员的笔记本永远是空白，即

traces(STOP) = {< >} □

X2 在接受一枚硬币后就毁坏的机器只能有两个迹

traces(coin → STOP) = {< >, <coin>} □

X3 只是滴答地走着的钟，它的迹是

traces(μX • tick → X) = {< >, <tick>, <tick, tick>,...}

= {tick}*

就象很多非常有趣的进程一样，尽管这个进程的每个迹都是有限的，其迹的集合却是无穷的。 □

X4 对一台简单自动售货机，其行为的迹为[67]

traces(μX • coin→ choc → X) ={ s | ∃ n • s ≤ <coin, choc>^{n} □

1.8.1 法则(Laws)

在这一节里，我们讲述如何使用目前已经阐述的符号系统来度量一个进程迹的集合。如前所述，进程 STOP 只有一个迹，并为

L1 traces(STOP) = {t | t = < >} = {< >}

空迹可以是(c → P)进程的一个迹，因为每个进程，到它开始实行第一个动作的时刻，< >都是它的行为的一个迹。(c → P)的每个非空迹都是以 c 开始，这个迹的尾部必须是 P 的一个可能的迹，故有

L2 traces(c → P) ={t | t = < > ∨ (t_0 = c ∧ t' ∈ traces(P))}

= {< >} ∪ {< c> ^ t | t ∈ traces(P)}[68]

对初始事件有多种选择的进程的行为的迹，一定是下面几种可能的迹之一

L3 traces(c → P | d → Q) =

{ t | t = < > ∨ (t_0 = c ∧ t' ∈ traces(P)) ∨(t_0 =d ∧ t' ∈ traces(Q)) }[69]

---

[67] 自动售货机不理解什么是一次交易完成。所以迹只是一种单纯的记录，可以包括<coin，choc，coin>。我们通过前缀算子 ≤来精确定义。s ≤ <coin，choc>^{n}

[68] 要么是最开始任何事件都还没有触发是的空迹状态，或者是以事件 c 为最开始的事件，然后跟着的是属于进程 P 的迹。

把上面三个法则(STOP, 选择，多项选择)归结为一个通用选择算子，可以归纳为

L4 traces(x : b → P(x)) ={ t | t = < > ∨ (t_0 ∈ B ∧ t' ∈ traces(P(t_0))) }[70]

对递归定义的进程来说，要定义其迹的集合要略微困难一些。一个递归定义的进程是下面方程的唯一解。

X = F(X)

首先，我们用归纳法定义函数 F 的迭代式

$F^0(X) = X$

$F^{n+1}(X) = F(F^n(X))$[71]

$= F^n(F(X))$

$=F(...(F(F(X)))...)$

然后，假设 F 是卫式，我们可定义

L5 traces(μX : A • F(X)) = $\cup_{n\geq 0}$ traces(F^n($STOP_A$))

例子

X1 回忆一下进程 RUN_A 在 1.1.3 节 X8 中被定义为

μX：A • F(X)

这里 F(X) = (x: a → X)，我们要证明

traces(RUNA) = A*

证明 A*=∪ {s|s ∈ A* ∧ (#s ≤n)} [72]

我们通过归纳发来证明。

1. traces($STOP_A$)

= {<>}

= {s|s ∈ A* ∧ #s≤0} [73]

2. traces(F^{n+1}($STOP_A$)

= traces(x：A → F^n($STOP_A$)) 根据 F 和 F^{n+1}的定义

= {t|t = <> ∨ (t_0 ∈ A ∧ t´∈ traces(F^n($STOP_A$)))} 根据 L4

= {t|t = <> ∨ (t_0 ∈ A ∧ (t´ ∈ A* ∧ #t´ ≤n))} 根据归纳假设[74]

---

[69] (t_0 = c ∧ t' ∈ traces(P)) 代表了选择一"c → P"的进程走向。(t_0 =d ∧ t' ∈ traces(Q))代表了选择一"d→ P"的进程走向。两个选择必居其一。

[70] 逻辑其实很简单，但数学表达的很精巧。进程的迹是根据最开始的事件选择的不同，响应的映射到不同的行为函数上。另外，t = t_0^t'，形成该进程完整的迹。

[71] 这里需要明确的是 n 不为 0。F^1(X)不能递归为 F(F^0(X))。进程一旦开始接受事件触发就进入正常的模式了。

[72]这个例子的证明与 Tony Hoare 1988 年的版本有一些文字上的变化。

[73]这里面蕴含了，当 n=0 时，即一个进程还不接受任何 event 的时候，其行为是 STOP 进程。

$= \{t \mid t = <> \vee (t_0 \in A \wedge t' \in A^*)) \wedge \#t \leq n+1\}$ #的特性

$= \{t \mid t \in A^* \wedge \#t \leq n+1\}$ 1.6.4 节 L4

□

X2 我们要证明 1.8 节中的 X4，即

$traces(VMS) = U_{n\geq 0}\{s \mid s \leq <coin, choc>^n$

证明 我们假设迭代为 n 的时候成立。

$traces(F^n(VMS)) = \{ t \mid t \leq <coin, choc>^n \}$

这里 F(X)=(coin → choc → X)

(1) $traces(STOP) = \{<>\} = \{s \mid s \leq <coin, choc>^0 \}$ 1.6.1 节 L6

(2) $traces(coin \rightarrow choc \rightarrow F^n(STOP))$

$= \{< >, <coin>\} \cup \{<coin, choc>^\wedge t \mid t \in traces\{F^n(STOP))\}$ L2

$= \{< >, <coin>\} \cup \{<coin, choc>^\wedge t \mid t \leq <coin，choc>^n\}$ 归纳假设

$= \{s \mid s = < > \vee s = < coin > \vee$

$\exists\ t \bullet s = <coin, choc>^\wedge t \wedge t \leq <coin, choc>^n\}$

$= \{s \mid s \leq <coin，choc>^{n+1}\}$

再由 L5 即得结论。 □

在 1.5 节中，我们阐述了迹是一个符号序列，这个序列记录了进程 P 到某一时刻为止已执行过的事件。其中< >是任何一个进程在开始执行第一个事件之前的迹[75]。另外，如果($s^\wedge t$)是进程到某一时刻的迹，则 s 一定是这个进程到前面某一时刻为止的一个迹。最后，发生的每一事件一定在进程的字母表内。这三个事实我们用法则正式给出。

L6 $< > \in traces(P)$

L7 $s^\wedge t \in traces(P) \rightarrow s \in traces(P)$

L8 $traces(P) \subseteq (\alpha P)^*$

进程的迹和进程行为的树状示意图是紧密相关的，对树上的任何节点来说，进程走到该节点时刻行为的迹就是它从树根到该节点沿途遇到的标记的序列。譬如，在图 1.1 中所示的 VMC 树形图上，与由树根到那个黑节点的路径对应的迹就是

---

[74] 归纳假设 $(t_0 \in A \wedge t' \in traces(F^n(STOP_A))) = (t_0 \in A \wedge (t' \in A^* \wedge \#t' \leqslant n))$

[75] 有点类似宇宙大爆炸之前的时间为 0 的状态一切都是静止的。从另外一个角度来考察，CSP 应该定义一个 INIT 的进程，其行为等价于 STOP 和后面定义的 SKIP。都表达一个进程什么也不响应的一种状态。但如果在这里 CSP 用 STOP 来描述进程最开始的迭代似乎有点不妥当。

<in2p, small, outlp >

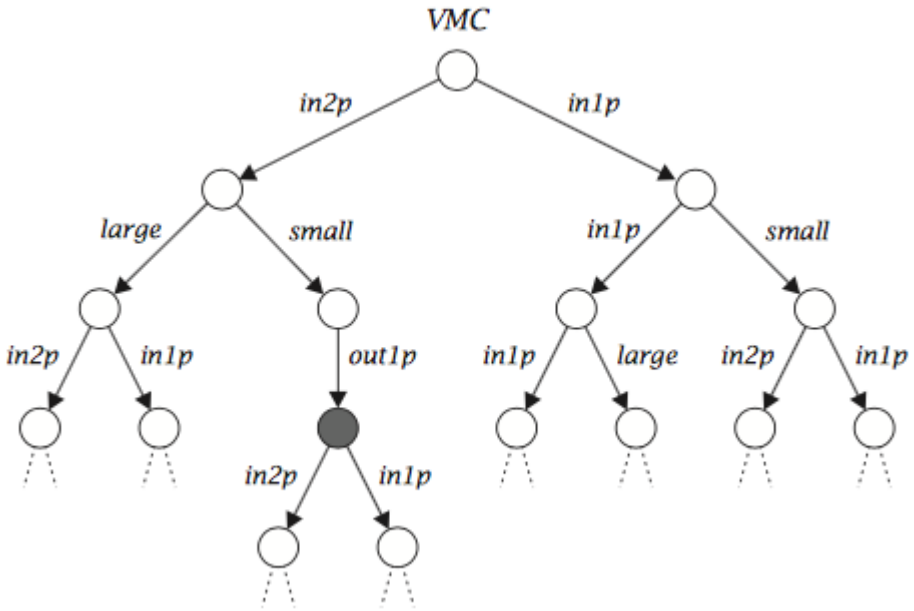


图 1.1

很显然，行为树上任意一条路径的所有子路径仍然是这棵树上的路径上述 L7 形式化的说明了这一点。空迹定义了由树根到其本身的空路径，即 L6[76]。进程所有的迹就是由树根引到节点的所有路径的集合。

反之，因为从每个节点引出的分支都用不同事件做了标记，所以进程的每个迹也就唯一地确定了由树根到某一具体节点的一条路径。这样，任何一个满足 L6 和 L7 的迹的集合组成了树形图的方便的数学表示式，在该类树形图中由一个节点不会引出两个标记相同的分支。

1.8.2 实现(Implementation)

假设进程已经通过 LISP 函数 P 来实现实了，这里 s 为此进程的一个迹。则通过下面的函数可以来验证是否 s 是 P 的一个迹

[76] 在定义递归函数的迹时，L5 的 $traces(\mu X : A \bullet F(X)) = \cup_{n\geq 0} traces(F^n(STOP_A))$ 通过 STOP 来表达最开始时的空迹，容易让人产生歧义。

istrace(s, P) = if s = NIL then

true

else if P(car(s)) = "BLEEP then

false

else

istrace(cdr(s),P(car(s)))[77]

由于 s 是有限的，所以这个递归在处理完进程 P 行为的有限步骤之后，会终止。正是因为我们不对进程作无穷的探究，我们才能把进程定义为一个无穷的对象，也就是说，进程是个函数，其结果仍为一个函数，这个结果函数的结果还是个函数.....

1.8.3 后继(After)

如果 s ∈ traces(P)，则

P/s (P 中 s 的后继)

是一个进程，这个进程表示 P 在执行完迹 s 所记录的所有动作后的行为。如果 s 不是 P 的一个迹，则(P/s)无意义。

例子

X1 (VMS /<coin>) = (choc → VMS) □

X2 (VMS / <coin, choc>) = VMS □

X3 (VMC /<(inlp >[3]) =STOP □

X4 为避免有 VMCRED (1.1.3 X5, X6)可能引起的损失，售货机的主人决定自己吃第一块巧克力，则有

(VMCRED /<choc>) = VMS2 □

在进程 P 的树形图上(如图 1.1)，(P/s)表示一棵完整的子树，子树的根就是用 s 所标记的路径的终点。因此，图 1.1 中黑色节点以下的子树就记为

VMC / <in2p, small, outlp>

以下几项法则将说明算子/的含义。什么事都还没做的进程不会有任何变化，因此

[77] 此处与 Tony Hare 1988 年的版本有点小区别。在 1988 年的书里直接用了 CSP 的符号表示一个迹的首元素和尾元素。在 LISP 里，用 Car 和 Cdr 来分别表示。The car of a list is, quite simply, the first item in the list. Thus the car of the list (rose violet daisy buttercup) is rose. The cdr of a list is the rest of the list, that is, the cdr function returns the part of the list that follows the first item.

L1 P/< >=P

执行完 s^t 后的迹，与跟 P/s 执行完再过滤 t 的行为完全一样，即

L2 P/(s^t) = (P/s)/t

执行完单个事件 c 后，进程的行为就要由这个初始选择来确定了，即

L3 (x:B → P(x)) / <c> = P(c) 假设 c ∈ B

下面的这条推论说明/ <c>是前缀算子 c → P 的逆运算。

L3A (c → P) / <c> = P

(P/s)的迹定义为

L4 traces(P / s) = {t | s^t ∈ traces(P)}　假设 s ∈ traces(P)

要证明进程 P 永远不停止动作，只要能对所有 s ∈ traces(P)

证明 P/s≠STOP 就够了[78]。我们想讨论的进程的另一个性质就是循环性：如果在任何情况下，进程 P 都能回到它的 z 最初始状态，即

∀ s : traces(P) • ∃ t • (P / (s^t)= P)

我们就称这个进程 P 是循环的。STOP 明显具备循环属性；但任何其它循环进程都是永远不停止动作的。

例子

X5 以下进程是循环的(1.1.3 节中 X8, 1.1.2 节中 X2,1.1.3 节中 X3,1.1.4 节中 X2)，即

RUN_A, VMS, (choc → VMS), VMCT, CT_7 □

X6 以下进程不是循环的，因为无法使它们回到初始状态(1.1.2 节中 X2, 1.1.3 节中的 X3，1.1.3 节中 X2)，即

(coin → VMS), (choc → VMCT), (around → CT_7)

譬如，在(choc → VMCT)的初始状态下，只能拿到巧克力，但是以后，choc 和 toffee 总是同时可供选择；因此这些后续状态就不会与初始状态相同了[79]。 □

注意：如果在递归定义的进程里使用/，会使其可能失去卫式特性[80]，并由此导致递归方程有多解。例如

X= (a → (X / < a >))

不是卫式，任何形式为

a → P

[78]永远抵达不到 STOP 状态，例如递归进程。

[79] VMCT = μX • coin → (choc → X|toffee→ X)

[80] 只有 Guarded 的卫式方程才具备唯一解。

的进程都是它的解，其中 P 可为任一进程。

证明 (a → ((a → P) / <a>)) =(a → P)[81]　　根据 L3A

由于这个原因，我们在递归进程的定义中不使用算子/。

1.9 迹的其它运算(More operations on traces)

这节主要讲述一下迹的其它运算。这部分目前可以跳过不看，在以后章节中如果用到这些运算时，我们会注明相关出处。

1.9.1 符号变换(Change of symbol)

假设函数 f 为集合 A 到集合 B 的符号映射。由 f 我们还可导出一个由 A*至 B*的新函数 f*，即用 f 分别作用于 A*中符号序列的每个元素，从而将 A*的符号序列对应为 B*中的一个序列。例如，设 double 为一个函数，使其整数自变量增为两倍，

double*(< 1, 5, 3, 1 >) = < 2, 10, 6, 2>[82]

标星号的函数很显然是服从分配律的，因而也是严格的，故有

L1 f*(< >) = < >

L2 f*(< x >) = <f(x)>

L3 f*(s^t) = f*(s) ^ f*(t)

其他明显的法则

L4 f*(s) = <f(s_0)>　　若 s ≠ < >

L5 #f*(s) = #s

下述一个法则貌似很“显然”，却其实并不成立

f *(s ↾ A) = f*(s) ↾f(A)

这里 f (A) = { f (x) | x ∈ A}。

我们举一个最简单的反例，有这样

一个函数 f，满足

f(b) = f(c) = c　　这里 b≠c，于是有

f*(<b> ↾ {c})　　由于 b≠c

= f*(<>)

[81] X= (a → (X / < a >))，代入 a → P 得到 a → P= (a → (a → P / < a >)。通过 L3A，得到 a → P = a → P。因此 P 可以为任何进程。不存在唯一解。

[82] 把一个迹的符号序列都相应的换掉。

= <>　　L1

≠ <c>

= <c>↾{c}

= f*(<c>)↾f({c})[83]　　由于 f(c) = c

然而，当函数 f 是 1-1 对应(单射)函数时，这项法则是成立的，既有

L6 f*(s↾A) =f* (s)↾f(A)　　当 f 为单射函数时

1.9.2 连接(Catenation)

设 s 为一序列，这个序列的每个元素本身还是一序列。则将所有元素按原序连接起来的结果记为^/s，例如

^/< <1,3>, < >, <7> > = <1,3> ^ < > ^ <7>

= <1, 3, 7>

这个算子满足分配律

L1 ^/< > = < >

L2 ^/<s> = s

L3 ^/(s^t) = (^/s) ^ (^/t)

1.9.3 交叉(Interleaving)

如果一序列 s 可被拆成一系列的子序列，并且这些字序列交替地为序列 t 和 u 的子序列，我们就说 s 是 t 和 u 的交叉。例如

s = <1, 6, 3, 1, 5, 4, 2, 7>

就是有 t 和 u 交叉而构成，这里

t=<1, 6, 5, 2, 7> 和 u = <3, 1, 4>

交叉算子的递归定义可由下述法则给出

L1 < > interleaves (t, u) ≡ (t = < > ∧ u=< >)

L2 s interleaves (t, u) ≡ s interleaves (u, t)

L3 ((x)^s) interleaves (t, u) ≡ (t ≠ <> ∧ t_0 = x ∧ s interleaves (t', u)) ∨

(u ≠ < > ∧ u_0 = x ∧ s interleaves (t, u'))

[83] f*(<c>)↾f({c}) = f*(<b>)↾f({c})。因此 f*(<b>↾{c}) ≠ f*(<b>)↾f({c})。这个反证可以这样理解：在有过滤算子的场景下，函数在过滤后和过滤前应用在一个迹上的结果可能有重要区别。因为函数的使用可以使得一个过滤算子失去效果。

1.9.4 下标(Subscription)

如果 0≤i＜#s，则我们用惯用的记号 s[i]，表示序列 s 的第 i 个元素，其定义见 L1。

L1 s[0] = s_0 ∧ s[i+1] = s′[i]　　若 s≠< >

L2 (f* (s)) [i]=f(s[i])　　若 i＜#s

1.9.5 逆置(Reversal)

设 s 为一序列，则$\underline{\underline{s}}$为 s 的逆置，即将 s 的元素顺序颠倒过来。例如

$\overline{3,5,37}$ = <37, 5, 3>

逆置的完整定义由以下法则给出

L1 $\overline{<>}$ = <>

L2 <$\bar{x}$> = <x>

L3 <$\overline{s\hat{}t}$> = $\bar{t}\hat{}\bar{s}$

逆置还具备几个简单的代数性质，包括

L4 $\bar{\bar{s}}$ = s

逆置的其它性质我们留给读者去推导。还有一个很有用的事实是，$\overline{s0}$是序列 s 的最后一个元素，且一般有

L5 $\bar{s}$[i] = s[#s - i +1]

1.9.6 挑选(Selection)

设 s 为一个二元序列，我们把 s 中所有第一个元素为 x 的元素对挑选出来，然后用各自的第二个元素取代这些元素对，所得结果定义为 s↓x。元素对的元素之间我们用一个黑点隔开。这样如果

s = <a，7，b，9，a，8，c.0 >

则　s↓a = <7,8 >

且　s↓d = < >

L1 < > ↓ x = < >

L2 (<y.z>^t) ↓ x = t ↓ x　　若 y≠x

L3 (<x.z>^t) ↓ x = < z> ^ (t ↓ x)

如果 s 不是二元序列，则 s↓a 表示 a 在 s 中出现的次数(如 1.6.6 节中定义)。

1.9.7 组合(Composition)

假设符号√表示进程的成功终止，则这个符号只能出现在迹的结尾[84]。设 t 为一个迹，它记录的是 s 已经终止后才开始的事件序列。s 和 t 的组合记为(s;t)。如果√没有在 s 中出现，则 t 就无法开始，即

L1 s ; t = s　若 ¬(< √> in s)

如果√在 s 的结尾出现了，就将它去掉，把 t 附加到 s 的剩余的部分，即

L2 s^<√ > ; t = s^t　　若 ¬(< √> in s)

符号√有点类似胶水，把 s 和 t 粘到一起。没有这个胶水，t 就粘不上去。如果符号√出现在一个迹的中间(是不正确的)，为了规则的完整性，我们规定符号√之后的序列都不参与组合，舍弃掉。即

L2A (s^⟨√⟩^u) ; t = s^t if ¬(⟨√⟩ in s)

这大家不太熟悉的组合算子具备几条常见的代数性质的。象连接一样，它是满足结合的。与连接不同的是，它不论对它的第一个变量还是第二个自变量都是单调的。而且组合算子对第一个自变量是严格的，并以<√>为其左单位元，即

L3 s; (t ; u) = (s ; t) ; u

L4A s ≤ t ⇒ ((u ; s) ≤ (u ; t))

L4B s ≤ t ⇒ ((u ; s) ≤ (t ; u))

L5 < >； t = t

L6 <√ >； t = t

如果√仅在迹的结尾出现，那么<√ >也是迹的右单位元，既有

L7 s；<√ > = s　　若 ¬(<√> in (s)′)

1.10 规约(Specifications)

一般说来，某项产品的功能描述说明这个产品的预期的行为。这类说明是包含一些自由变量的谓词断言[85]，每个自由变量表示这个产品行为的某个可观察到的方面。例如，有这样一个电子放大器的描述，其电压输入范围为 1 伏，放大约为 10 倍，表示成谓词就是

AMPIO = (0≤v≤1⇒|v′-10×v| ≤1)

[84] 这个事件对应的是第 5 章顺序进程中介绍的“SKIP”进程，表示一个进程成功结束。

[85] Predicate 的意思。本书里的都应该是一介逻辑的谓词演算。

在描述中，谓词中的 v 理解为输入电压，v′为输出电压。在科学和工程中使用数学时，很基本的一点就是要理解变量的含义。

对进程而言，能观察到的和进程行为最直接有关的，就是到某一给定时刻为止发生的事件的迹。我们用特殊变量记号 tr 表示所描述的进程的一个任意迹，这和前一段中我们用记号 v 和 v′记载观察到的电压的任意变化的方法是一样的。

例子

X1 自动售货机的主人不想亏本，他就规定送出巧克力的块数不能超过投入的硬币数

$$NOLOSS = (\#(tr \upharpoonright \{choc\}) \leq \#(tr \upharpoonright \{coin\}))$$ □

以后我们用缩写形式

$$tr \downarrow c = \#(tr \upharpoonright \{c\})$$

表示符号 c 在 tr 中出现的次数(见 1.6.6 节)

X2 顾客也希望自动售货机公平合理，投入多少硬币就能得到多少巧克力

$$FAIR1 = ((tr \downarrow coin) \leq (tr \downarrow choc) + 1)$$ □

X3 简单自动售货机的生产就必须同时满足机器主人和顾客的要求

$$VMSPEC = NOLOSS \wedge FAIR1$$

$$= (0 \leq ((tr \downarrow coin) — (tr \downarrow choc)) \leq 1)$$ □

X4 修改复杂自动售货机的描述，不允许它连续接受三枚硬币

$$VMCFIX = (\neg \langle inlp \rangle^3 \text{ in } tr)$$ □

X5 改进后的机器的描述为

$$MENDVMC = (tr \in traces(VMC) \wedge VMCFIX)$$ □

X6 VMS2(1.1.3 节 X6)的描述

$$0 \leq ((tr \downarrow coin) — (tr \downarrow choc)) \leq 2$$ □

1.10.1 满足(Satisfaction)

设 P 为满足描述 S 的产品，我们说 P 满足 S，缩写为

$$P \text{ sat } S$$

其含义是，可观察到的 P 的行为都被 S 所描述的行为刻画了；换句话说，只要 S 中的变量取的是由观察 P 得到的值时，S 就为真，或更形式地记为 $\forall tr \bullet tr \in traces(P) \Rightarrow S$。譬如，下列表格给出对放大器特性的观察结果

	1 2 3 4 5
V	0 .5 .5 2 .1
V'	0 5 4 1 3

所有这些观察结果除了最后一栏外都满足 AMP10。第二和第三栏说明放大器的输出并不是完全由输入决定的。第四栏表明，如果输入电压超出了指定范围，输出电压可以是任意值，但这不算是违背描述。(在这个简单的例子里，我们排除过高电压会使产品击穿的可能性。)

下面的一些法则给出"满足"关系的几个最一般的性质。True 不对产品施加任何限制，因此所有产品都满足这一要求；甚至是坏掉的产品也能满足这一无意义的、无需求的描述。即

L1 P sat true

如果一产品同时满足两种不同的规约，则也满足它们的并合，即

L2A 如果 P sat S

并且 P sat T

则有 P sat (S ∧ T)

法则 L2A 可推广到无穷个描述的逻辑与，即推广到全称量词。设 S(n)为含变量 n 的谓词

L2 当 P 不随 n 变化时

如果∀ n • (P sat S(n))

则 P sat (∀n • S(n))[86]

如果描述 S 从逻辑上讲蕴含了另一描述 T，则由 S 刻画的观察也可用 T 来刻画。于是每个满足 S 的产品也一定满足较弱的描述 T，即

L3 如果 P sat S

且 S ⇒ T

[86] P sat (S(0) ∧ S(1)... ∧ S(n))。证明很简单，true ∧true ... ∧true = true。

则 P sat T

有了这项法则，我们有时可将证明串起来；如果 S⇒T，我们将

P sat S

⇒T

作为下面完整证明的缩写形式。

P sat S

S ⇒T

P sat T　　由 L3

以上给出的法则及其解释对所有各类产品及各类描述都适用。在下一节里，我们将给出适用于进程的其它法则。

1.10.2 证明(Proofs)

在设计产品时，设计人员的责任就是确保设计的产品能满足相关要求，要做到这一点，可能使用有关数学分支中的论证方法，譬如几何学或微积分学中的方法。在这一节里，我们要给出的一组法则，从而可运用数学中的论证方法来确保进程 P 满足它的规约 S。

我们有时把一个规约写作 S(tr)，以表示一个规约通常都含有自由变量 tr[87]。然而我们显示的标注变量 tr 的真正原因在于要说明 tr 可由别的复杂表达式代换，例如 S(tr")。必须强调，S 和 S(tr)中除了有变量 tr 之外，还可有其它的自由变量。

不论怎样观察进程 STOP，它的迹永远是空迹，因为这个进程什么也不做，故有

L4A STOP sat (tr = < >)

进程(c → P)的迹一开始是空的。每个后续的迹均以 c 开始，其尾部又是 P 的一个迹。所以这个尾部一定是由 P 的任何一个描述所刻画的，即

L4B 如果 P sat S(tr)

则 (c → P) sat (tr = < > ∨ (tr_0 = c ∧ S(tr')))

这个法则的推论可以解决双前缀的问题。

L4C 如果 P sat S(tr)

则 (c → d → P) sat (tr ≤ < c, d> ∨ (tr ≥ < c, d> ∧ S(tr")))[88]

---

[87] 一介谓词，first order logic。

[88] tr ≤ < c, d>的含义是 tr 可以是空，{c}，或者{c, d}。其演算里包含了<>空迹，可参阅 1.6.5 关于 Ordering 的 L1。空迹是任何迹的前缀。(tr ≥< c, d>∧ S(tr")))的演算形式化描述了：tr 如果

二元选择的情况与前缀类似，区别仅在于，迹可以以两个供选择事件中的任意一个开始，而迹的尾部必须由被选中分支的描述来刻画，故有

L4D 如果 P sat S(tr)

且 Q sat T(tr)

则 $(c \rightarrow P \mid d \rightarrow Q)$ sat $(tr = <> \vee (tr_0 = c \wedge S(tr')) \vee (tr_0 = d \wedge T(tr')))$[89]

以上给出的几条法则其实都是通用法则 L4 的特例

L4 如果$\forall x \in B \bullet (P(x)$ sat $S(tr, x))$[90]

则 $(x : b \rightarrow P(x))$ sat $(tr = <> \vee (tr_0 \in B \wedge S(tr', tr_0)))$

后继算子的法则简单得出奇。如果 tr 是(P/s)的迹，则 $s^\frown tr$ 是 P 的迹，因此 $s^\frown tr$ 必须由 P 所满足的描述来刻画，即

L5 如果 P sat S(tr)

且有 $s \in traces(P)$

则(P/s) sat $S(s^\frown tr)$[91]

最后，我们还需要一条法则建立递归进程的正确性。

L6 如果 F(X)是卫式表达式，

且 STOP sat S

且$(X \text{ sat } S) \Rightarrow (F(X) \text{ sat } S)$

则$(\mu X.F(X))$ sat S

这项法则的前提条件保证了可用归纳法证明

$F^n(STOP)$ sat S　　对一切 n[92]

长度超过 cd 之后，去除了前面两个字母之后的迹 tr"满足 S。这一点也很直接。因为如果(c → d → P)，其后面的迹是 P 的迹。而 P 的迹是满足 S 的。综合起来，(c → d → P)一定满足(tr ≤ < c, d> ∨ (tr ≥< c, d>∧ S(tr")))的两个选择之一。

[89] 通过 L4B，可以非常容易的推出这个结论。

[90] S(tr, x)表达了在多重选择的情况下的表达方式。例如，当一个选择是 x 的时候，其相应的规约是 tr。

[91] 因为是 P sat S(tr)，所以(P/s)的迹需要补一个 s 在 tr 前面，$s^\frown tr$ 才能合成一个满足 P 的迹去满足 P。即 sat $S(s^\frown tr)$。

[92] 证明：1. 对递归函数 μX.F(X)，STOP 属于其迹。STOP sat S 成立。2. 假设 $F^n(STOP)$ sat S 成立。$F^{n+1}(STOP) = F(F^n(STOP))$。因为已知$(X \text{ sat } S) \Rightarrow (F(X) \text{ sat } S))$，因此 $F^{n+1}(STOP)$ sat S。 因此，根据归纳法，$F^n(STOP)$ sat S，对一切 n。

由于 F(X)是卫式，F^n(STOP)至少完全描述了 μX.F(X)的前 n 步的行为。故 μX.F(X)的每个迹都是某个 n 的 F^n(STOP)的迹。所以这个迹必须满足 F^n(STOP)的描述，而这个描述对所有 n 来说都是 S。我们将在 2.8 节中用数学理论给出更形式的证明。

例子

X1 我们欲证(1.1.2 节 X2, 1.10 节 X3)

VMS sat VMSPEC

证明

1. STOP sat tr = < > L4A

$\Rightarrow 0 \le (tr\downarrow coin - tr\downarrow choc) \le 1$

$(< >\downarrow coin) = (< >\downarrow choc) = 0$

这个结论中隐含着对法则 L3 的应用。

2. 假设 X sat$(0 \le tr\downarrow coin) - (tr\downarrow choc)) \le 1$

于是 $(coin \rightarrow choc \rightarrow X)$ sat $(tr \le <coin, choc>)$

$\vee$

$((tr \ge <coin, choc> \wedge 0 \le ((tr'' \downarrow coin) - (tr'' \downarrow choc) \le 1)$[93] L4C

$\Rightarrow 0 \le ((tr \downarrow coin) - (tr \downarrow choc)) \le 1$

因为$< >\downarrow coin = < >\downarrow choc = < coin>\downarrow choc = 0$

并且$< coin>\downarrow coin = (<coin, choc>\downarrow coin) = <coin, choc>\downarrow choc = 1$

并且 $tr \ge <coin, choc> \Rightarrow (tr\downarrow coin = tr''\downarrow coin+1 \wedge tr\downarrow choc = tr''\downarrow choc + 1)$[94]

应用 L3，再应用 L6 即得证。

进程 P 满足它的描述并不一定就意味着是可用的。譬如，因为有

$tr = < > \Rightarrow 0 \le (tr\downarrow coin-tr\downarrow choc) \le 1$

应用 L3 和 L4，可证

STOP sat $0 \le (tr\downarrow coin - tr\downarrow choc) \le 1$

---

[93] 通过 L4C，基于 X sat$(0 \leqslant tr\downarrow coin)-(tr\downarrow choc)) \leqslant 1$，可以扩展定义$(coin \rightarrow choc \rightarrow X)$的规约。

[94] 因为 $tr\downarrow coin = tr''\downarrow coin+1 \wedge tr\downarrow choc = tr''\downarrow choc+1$ 为真，所以，$tr\downarrow coin - tr\downarrow choc = tr''\downarrow coin - tr''\downarrow choc$(原因非常直接：前面的序列是一个 coin 和一个 choc)。根据假设 X sat$(0 \leqslant tr\downarrow coin)-(tr\downarrow choc)) \leqslant 1$。因此递归$(coin \rightarrow choc \rightarrow X)$之后的迹仍然满足 $0 \leqslant ((tr \downarrow coin) - (tr \downarrow choc)) \leqslant 1$。

然而 STOP 做为自动售货机可不够，它对主人和顾客都无用。STOP 自然不会做错任何事情；但是办法太消极，什么都不做。也正是由于这个原因，STOP 能满足任何进程所满足的描述。

幸运的是，很容易地可以推理证明 VMS 永远不会停止工作。其实任何进程，只要它是仅有前缀、选择以及卫式递归式定义的，就不会停止动作。设计一个能停止的进程只有一个办法，就是把 STOP 显示的写上去，或者(x：b → P(x))，这里 B 为空集。只要避开了这类初级错误，我们就能保证写出来的进程就是永远不停止的。然后，在下一章中引入并发性的问题之后[95]，这些简单的防御措施就无法满足不停机的要求了。在第三章的 3.7 节中我们将给出描述和证明某个进程永远不会停止的更一般方法。

[95] 并发(Concurrency)会导致两个貌似都不错(例如，不停机)的进程，互相死锁，并导致整体系统停机 STOP。

第二章　并发(CONCURRENCY)

2.1 引言(Introduction)

进程是通过刻画其全部的潜在行为来定义的。而且经常要在几种不同动作之间做出选择，譬如说，是给 VMC(1.1.3 节 X4)投入一枚大点的硬币呢，还是投入一枚小一点的。每当出现这种情况时，到底会发生哪个事件则完全由进程所处的环境控制。例如，总是由顾客来决定到底想投入什么样的硬币，值得庆幸的是，我们定义进程的这套符号系统记，也可用来定义一个进程的环境的行为，从而将进程的环境也刻画成一个进程，这样，我们就可以把进程连同它所在的环境合到一起，作为一个完整的系统，对其行为加以探讨和研究；在进程及其环境并发的运行过程中，它们各自动作着，且又在交互作用着。这个完整的系统也应当被看作是一个进程，它的行为可由组成它的各个进程的行为决定；而且这个系统也有可能被置于一个更大、更广的环境中去。其实，最好是忘掉进程、环境和系统三者相互之间的区别，它们统统都是进程，这些进程的行为都可以用一种简单而又统一的方式规定、刻画、记录和分析。

2.2 交互作用(Interaction)

当把两个进程放到一起，让它们并行运行时，我们一般是期望它们交互作用。这些交付行为可以被看作是要求这两个进程同时参予的事件[96]。暂时我们还是把注意力集中到这类同时参予的事件上，先不管其它事件。因此我们先假设两个进程的字母表相同。于是每个实际发生的事件，都必须是每个进程独立行为中的一个可能事件。比方说，只有当顾客需要巧克力，而且自动售货机也提供巧克力的时候，巧克力才能送出来[97]。假设 P 和 Q 是两个字母表相同的进程，我们引入

$$P \parallel Q$$

[96] 同时参与是并发算子的本质。是指并发进程要从开始第一个事件触发开始有共同的响应的行为(迹)，否则观察者无法记录这个组合进程下一步是做什么，进程的子进程无法达成共识。在第三章里会引入和接受关于非确定性的算子，例如，两个子进程可以按照自己支持的，不同于另外一个子进程的行为来出来外界环境和之后的处理流程。

[97] 这里蕴含的意思是环境(顾客)与售货机(进程)都需要能够处理“巧克力”这个事件，顾客与售货机才能互动协调起来共同相应“巧克力”这个事件。

表示一个进程，其行为就是 P 和 Q 构成的整个系统的行为，P 和 Q 的交互作用按上述说明为严格同步的[98]。

例子

X1 有位贪心的顾客想不付钱就拿到一块太妃糖或一块巧克力。只在他这种欲望得不到满足时，他才不情愿的付一枚硬币；一旦付了钱后，他就非要块巧克力不可了。即有

GRCUST = (toffee → GRCUST
| choc → GRCUST
| coin → choc → GRCUST)

当这位顾客遇到 VMCT(1.1.3 节 X3)时[99]，就不能得逞了，因为这台机器不允许先尝后付钱。同时，因为这个人付钱之后只要巧克力，所以 VMCT 也就不会给出太妃糖来[100]。因此有(GRCUST ‖ VMCT) = μX. (coin → choc → X)[101]

这个例子说明，不用并发算子‖，也可以把两个子进程组合成的进程描绘成一个单一的简单的进程[102]。

X2 一位傻乎乎的顾客想买一块大饼干，于是就把硬币投入 VMC。他并没有注意自己投入是哪种硬币；可是不管怎么说，他非要块大饼干不可，他的行为模式可以形式化的描述为：

FOOLCUST = (in2p → large → FOOLCUST
|inlp → large → FOOLCUST)

可惜的是，自动售货机可不愿吃亏。这位顾客投入一枚小硬币，它才不会给出大饼干的，故

(FOOLCUST‖VMC) = μX • (in2p → large → X| inlp → STOP)[103]

伴随着 inlp 后发生的 STOP 被叫做死锁。尽管每个分进程都准备好进行下一步的动作，但它的各自的下一步动作都是不同的；由于这些进程无法统一意志，下一步就什

---

[98] 可以认为 P 是 Q 的环境。也可以认为 Q 是 P 的环境。互相同步，互相牵制，形成一个共同对外的系统。

[99] VMCT = μX • coin → (choc → X | toffee → X)。显然，这个售货机只支持先付钱，才有其他可能(糖或者巧克力)的行为。

[100] 两个进程(顾客和售货机)同步，最后达到一个一致对外的行为结果。所以，不可能有糖的选择，不是一个两个进程都共同存在的一个演化路径。

[101] 两个进程并发，是一种合作行为，不是各自为政的非确定性的随机行为。并发的结果是结果的迹在任意一个进程里都是一致的。从而，两个进程合成的进程对外体现的是一个唯一的，一致的行为和记录下来确定的迹。

[102] 也可以反过来思考：任何一个进程，也可以通过并发算子把两个子进程组合起来的行为来表示。

[103] VMC = (in2p → (large →VMC | small → outlp →VMC) | inlp → (small →VMC | inlp → (large → VMC| inlp → STOP)))。可以很容易的观察，当投入一个 inlp 的时候，进程 FoolCust 的行为和 VMC 的行为无法取得下一步的共同的步骤。因此，只能是 STOP。

么也不会发生了[104]。 □

以上例子的故事，违背了科学抽象性和客观性的标准。但切记一点，各种事件都特意定义成为一些中性的过度状态，从而可以由那些不懂得七情六欲的外星人观察和记录下来。这类外星人不懂吃饼干的乐趣，也不了解那位愚蠢的顾客徒然等待粮食时挨饿的痛苦。我们选择有关事件的字母表时有意地排除了这些内在的感情因素；但如果真有这种需要时，我们还可引入某类事件来模拟这种内在的状态变化，详情可参见 2.3 节中的 X1。

2.2.1 法则(Laws)

(P ‖ Q)的行为遵从的法则非常简单直接。第一条法则说明进程与其环境之间的逻辑对称性，即

L1 $P \| Q = Q \| P$[105]

下一个法则说明当三个进程组装在一起时，它们之间的排列顺序没有关系

L2 $P \| (Q \| R) = (P \| Q) \| R$[106]

第三，一个死锁进程会使整个组合的系统都死锁；但是一个进程如果与 $RUN_{\alpha P}$(1.1.3 节 X8)组合，结果不会受到影响。

L3A $P \| STOP_{\alpha P} = STOP_{\alpha P}$

L3B $P \| RUN_{\alpha P} = P$[107]

下一条法则说明一对进程或者同时执行同一动作，或者在执行第一个动作时无法取得一致意见，而处于死锁，即

L4A $(c \rightarrow P) \| (c \rightarrow Q) = (c \rightarrow (P \| Q))$[108]

L4B $(c \rightarrow P) \| (d \rightarrow Q) = STOP$[109] 若 $c \neq d$

这两条法则可以很容易地推广到两个进程都可以选择初始事件的情况；当把这两个进

---

[104] 假设这两个独立的进程，都通过一个共同的外部事件触发，例如，in1p，(这时观察者可以记下这个迹。)，但两个进程开始出现分歧，对 in1p 触发之后的处理不一致，因此导致符合进程无法再往下发展，成为 STOP 状态。

[105] 并发算子满足交换律

[106] 并发算子满足结合律

[107] 有点类似一个子集与一个无穷大的全集做过滤。显然过滤的结果是这个子集。

[108] 两个进程在对外的第一个 event 上，有共同的事件。因此可以往下走一步。外界观察者能够记录下来 c 作为 trace。不会出现记录不下来的现象。

[109] 对这个法则可以从两个方面来理解。1. 这两个进程组合起来的进程无法在第一个事件上取得共识，从而导致方程右侧的行为无法被观察者做任何记录，形成迹(trace)。2. 从 1.8 的法则 L6， 我们可以指导，< >空迹其实属于任何一个进程，例如，这个组合进程。因为这个组合进程的唯一的两个组成部分不可能达成对第一个事件的相应，因此，这个组合进程的行为只能是< >。或者说是一个空进程 STOP。

程组合到一起时，只有他们的公共选择部分才能保留下来，即

L4 $(x: a \rightarrow P(x)) \parallel (y: b \rightarrow Q(y)) = (z:(A \cap B) \rightarrow (P(z) \parallel Q(z)))$[110]

规则 L4 表明使用并发算子定义的系统可以由不使用并发算子的方式来刻画的。

例子

X1 设 $P = (a \rightarrow b \rightarrow P \mid b \rightarrow P)$

且 $Q = (a \rightarrow b \rightarrow Q \mid c \rightarrow Q)$

则 $(P \parallel Q) = a \rightarrow (b \rightarrow P) \parallel (b \rightarrow \mid c \rightarrow Q))$　　由 L4A

$= a \rightarrow (b \rightarrow (P \parallel Q))$

$= \mu X \cdot (a \rightarrow b \rightarrow X)$　　因为这个递归方程式是卫式

2.2.2 实现(Implementation)

显然组合算子‖的实现要以规则 L4 为基础

intersect(P, Q) = λ z •

if P (z) ="BLEEP ∨ Q(z) ="BLEEP then"BLEEP else intersect(P(z), Q(z))

2.2.3 迹(Traces)

由于(P ‖ Q)的每一个动作都要 P 和 Q 同时参与，则这类动作的每个序列必须同时是这两个运算对象的动作序列。由于这个原因，/s 对‖算子有分配律。

L1 $traces(P \parallel Q) = traces(P) \cap traces(Q)$[111]

L2 $(P \parallel Q)/s = (P/s) \parallel (Q/s)$[112]

2.3 并发性(Concurrency)

上节中讲述的算子也可以推广到运算对象 P 和 Q 的字母表不同的情形，即

$\alpha P \neq \alpha Q$

当把这样的两个进程组装到一起并发运行时，它们字母表中共有的事件就需要 P 和 Q 同时参加，其理由见 2.2[113]。然而，属于 αP 但不属于 αQ 的事件就跟 Q 无关系了，也没

---

[110] 要么是空集，组合后的进程是一个 STOP 进程，即虽然内部暗潮涌动(由两个或者多个子进程组成)，但什么也做不了，无法对外接受事件的触发，形成不了一致意见。

[111] 基于处理的每一步，两个进程共同的行为。

[112] 在迹 s 之后的迹。这里显然 s 是复合进程的迹。

[113] 共有的事件的含义是：对这些事情的处理是对外可观察到的，需要被记录成为 trace 的。因此，假设在某一事件之前，所有的动作序列都是符合并发算子的要求的。但是在这个当前事件上(事件属于各

能力去控制甚至过问这类事件。因此每当 P 在执行这类事件时，它们可以独立于 Q 发生。类似地，Q 也会执行只属于 αQ 而不属于 αP 的事件[114]。这样，只需将参加组合的各个进程的字母表简单地并到一起，就可得到整个组合系统逻辑上所有可能事件的集合。故

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$

这种运算对象的字母表不相同的算子不多见，而且运算结果又产生了另一个字母表。但在两个进程字母表相同的情况里，它们结合后字母表仍然不变，(P ‖ Q)的含义与上节所述的完全一样。

例子

X1 假设 αNOISYVM = {coin, choc, clink, clunk, toffee}

这里 clink 是硬币掉进自动售货机时的声音，clunk 是自动售货机做完一批交易后发出的声音。另外，这台响个不停的售货机里已经售完太妃糖了，其行为可以形式描述为：

$$NOISYVM = (coin \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow NOISYVM)$$

但是顾客就想要太妃糖；curse 是顾客拿不到太妃糖时的咒骂；后来他还是不得不拿一块巧克力。故

$$\alpha CUST = \{coin, choc, curse, toffee\}$$

$$CUST = (coin \rightarrow (toffee \rightarrow cUST \mid curse \rightarrow choc \rightarrow CUST))$$

这两个进程并发动作的结果就成了[115]

$$(NOISYVM \parallel CUST) = \mu X \bullet (coin \rightarrow (clink \rightarrow curs \rightarrow choc \rightarrow clunk \rightarrow X \mid curse \rightarrow clink \rightarrow choc \rightarrow clunk \rightarrow X))$$[116]

注意，clink 可能发生在 curse 之前，也可能在它之后。这两个事件甚至有可能同时发生，但记录它们的前后次序是无关紧要的[117]。

个进程的字母表)，处理的下一步如果属于共同的事件，则必须一致，否则观察者无法记录迹，类似系统不知道到底是做哪个事件。符号表里的事件的定义是一个进程必须对这个事件有所反应。

[114] 对任何一个只属于某一个进程 P 的事件，不需要其余那个进程 Q 行为的支持。这个进程 P 里事件的发生不会干扰到进程 Q 的行为。对作为记录复合并发进程(P ‖ Q)的迹的观察者，不会无所适从。

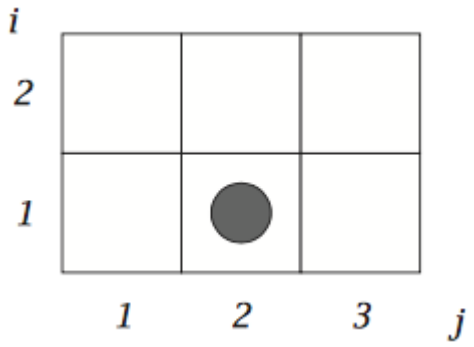
[115] 在这个复合的并发进程里，其中 clink，clunk 事件只会被 NOISYVM 进程所感知和触发(或者被触发)；curse 智慧被 CUST 客户进程所触发。

[116] 如果不考虑那些独立的事件，这两个进程复合的对外的行为，或者是可以被记录下来的迹是：μX • (coin → choc → X)。一个只接受硬币，吐出巧克力的行为。

[117] clink 和 curse 的事件发生不存在依赖关系。但都必须这 choc 之前。满足自己对应进程行为描述中事件的前后顺序。

还要注意的是，数学公式是无法表现顾客的感情的，他当然情愿要块太妃糖而不愿意说句难听的话。数学公式作为一种对现实的抽象，忽略人的感情因素，只去描述进程字母表的事件发生或不发生的可能性，不管人们是否喜欢这些事情。

X2



一个筹码从如上图所示的方格开始，这个筹码可在板上、下、左、右移动。

假设

$\alpha P = \{up, down\}$

$P = (up \rightarrow down \rightarrow P)$

$\alpha Q = \{left, right\}$

$Q = (right \rightarrow left \rightarrow Q \mid left \rightarrow Right \rightarrow Q)$

筹码的行为可定义为 $P \parallel Q$。

在此例中，P 和 Q 没有一个共同事件。所以，这个筹码的运动就是 P 和 Q 的动作的任意交叉。这种交叉如果不用并发性而用其它方法来刻画，就会很冗长。例如，设 R_{ij} 表示筹码由板上第 i 行、第 j 列的位置上出发的可能的行为，其中 $i \in \{1, 2\}, j \in \{1, 2, 3\}$。则有

$(P \parallel Q) = R_{12}$　其中

$R_{21} = (down \rightarrow R_{11} \mid right \rightarrow R_{22})$

$R_{11} = (up \rightarrow R_{21} \mid right \rightarrow R_{12})$

$R_{22} = (down \rightarrow R_{12} \mid lef \rightarrow R_{21} \mid right \rightarrow R_{23})$

$R_{12} = (up \rightarrow R_{22} \mid left \rightarrow R_{11} \mid right \rightarrow R_{13})$

$R_{23} = (down \rightarrow R_{13} \mid left \rightarrow R_{22})$

$R_{13} = (up \rightarrow R_{23} \mid left \rightarrow R_{12})$ □

2.3.1 法则(Laws)

前三条法则是对并发性的扩展，与交互作用的法则(2.2.1 节)非常类似，它们是

L1, 2 ‖既是对称的又是结合的

L3A $P \parallel STOP_{\alpha P} = STOP^{118}{}_{\alpha P}$

L3B $P \parallel RUN_{\alpha P} = P$

假设 $a \in ((\alpha P - \alpha Q), b \in (\alpha Q - \alpha P)$，且$\{c, d\} \subseteq ((\alpha P \cap \alpha Q)$。那么以下的几条法则说明，P 可单独执行事件 a，Q 可单独执行事件 b，但执行 c 和 d 时需要 P 和 Q 同时参加[119]。

L4A $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

L4B $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP^{120}$ 若 $c \neq d$

L5A $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))^{121}$

L5B $(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)^{122}$

L6 $(a \rightarrow P) \parallel (b \rightarrow Q) = a \rightarrow (P \parallel (b \rightarrow Q)) \mid b \rightarrow ((a \rightarrow P) \parallel Q))^{123}$

上述法则可推广到通用选择算子上。

L7 设 $P = (x: a \rightarrow P(x))$

且 $Q = (y: b \rightarrow Q(y))$

则 $(P \parallel Q) = (z: c \rightarrow P \parallel Q')$

这里 $C = (A \cap B) \cup (A - \alpha Q) \cup (B - \alpha P)$

且有 $P' = P(z)$[124] 如果 $z \in A$

$= P$[125] 否则

---

[118] 组合后的进程没法对字母表 αP 里的任何事件做出响应。

[119] 同时参加，一致，从而复合进程可以完整的体系一个不混乱的迹。

[120] 根据并发算子的定义，(c → P)进程能够处理的第一个非空的事件是 c；(d→ Q)能够处理的第一个非空的事件是 d；而且 c≠d。因此，((c → P) ‖ (d→ Q))这个进程陷入了混沌状态，没法往任何一个共同的行为方向发展，只能进入 STOP。

[121] a 是(a → P)进程或者说 P 进程独有的事件。整个复合进程的行为显然可以支持 a 的发生，而不需要考虑(c → Q)的同步并发问题。如果 a 发生并且记录下 a 之后，这个时候要“看”之后的 P 和(c → Q)了。因为 c 是 P 和 Q 都需要反应的事件。所以，未来的整体行为不好说，a 发生后，系统的行为只能是(P ‖ (c → Q))。1. 我们假设 P 是(d→ P')。我们可以很容易的理解，这个复合并发进程的迹最多能记下一个 a 事件，然后就会进入 STOP。原因是(d → P') ‖ (c → Q)的首发无法能够取得一致。整体系统陷入混乱(chaos)状态，只能 STOP。2. 如果 P 是(c → P')， 整个进程显然可以往下多发展一步，因为，(c → P') ‖ (c → Q) = c → (P' ‖ Q)。系统的行为，或者迹可以发展到<a, c>了。。。依此逻辑，可以一直推理下去。

[122] 与 L5A 是一个逻辑。

[123] a 与 b 是互相独立的可以交错的事件。所以整体组合进程的行为可能是 a 先发生，或者 b 先发生。值得注意的是，a → (P ‖ (b → Q))里的不能简单的推导出 b → (P ‖ Q)。原因是我们并不知道 P 的细节。如果 P 是类似(c → P')，根据 L5B，(P ‖ (b → Q)) = (c → P' ‖ (b → Q)) = b → (c → P' ‖ Q) = b → (P' ‖ Q)；但如果 P 还是(a → P')的模式，(P ‖ (b → Q)) = (a → P' ‖ (b → Q))，那么后续行为是 a → (P' ‖ (b → Q))|b → ((a → P') ‖ Q))。

[124] 当 C 属于 (A ∩ B) ∪ (A-αQ)的时候。

[125] 当 C 属于(B-αP)的时候。

$Q' = Q(z)$ 如果 $z \in B$

$= Q$ 否则

利用这些法则，可以把由并发算子定义的进程简化，重新定义，使其中不出现并发算子，如下例。

例子

X1 设

$\alpha P = \{a, c\}, P = (a \rightarrow c \rightarrow P)$

$\alpha Q = \{b, c\}, Q = (c \rightarrow b \rightarrow Q)$

于是 $P \| Q = (a \rightarrow c \rightarrow P) \| (c \rightarrow b \rightarrow Q)$ 由定义

$= a \rightarrow ((c \rightarrow P) \| (c \rightarrow b \rightarrow Q))$ 由 L5A

$= a \rightarrow c \rightarrow (P \| (b \rightarrow Q))$ 由 L4A...‡

而且 $P \| (b \rightarrow Q)$[126] $= (a \rightarrow (c \rightarrow P) \| (b \rightarrow Q)$

$| b \rightarrow (P \| Q))$ 由 L6

$= (a \rightarrow b \rightarrow ((c \rightarrow P) \| Q)$[127]

$| b \rightarrow (P \| Q))$ 由 L5B

$= (a \rightarrow b \rightarrow c \rightarrow (P \| (b \rightarrow Q))$[128] 由‡

$| b \rightarrow a \rightarrow c \rightarrow (P \| (b \rightarrow Q)))$ [129]

$= \mu X \bullet (a \rightarrow b \rightarrow c \rightarrow X)$ 因为是卫式

$| b \rightarrow a \rightarrow c \rightarrow X$

所以

$(P \| Q) = (a \rightarrow c \rightarrow \mu X \bullet (a \rightarrow b \rightarrow c \rightarrow X$

$| b \rightarrow a \rightarrow c \rightarrow X))$ 由‡

□

[126] 首先展开为(a → c → P) ‖ (b → Q)。因为 a，b 是独立的两个事件分别属于 αP 和 αQ，因此应用 L6 开始推导。

[127] 通过 L5B，(c → P) ‖ (b → Q)可以容易的推导为 b → ((c → P) ‖ Q。

[128] 开始化简(c → P) ‖ Q。把 Q = (c → b → Q)代入，得到(c → P) ‖ (c → b → Q)。因此，得到 c → (P ‖ (b → Q))。代入到上一步的方程式中，得到 a → b → c → (P ‖ (b → Q))。

[129] 把前面得到的 P ‖ Q =a → c → (P‖(b → Q))代入 b → (P ‖ Q)，得到 b → a → c → (P ‖ (b → Q))。

2.3.2 实现(Implementation)

算子‖的实现可直接由 L7 导出。我们可以把运算对象的字母表表示成符号的有限表 A 和 B。要检验一符号是否在表中，就用我们在 1.7 节中定义过的函数 ismember(x,A)

要实现(P ‖ Q)，可以通过调用函数 concurrent(P, αP, αQ, Q)

这个函数是这样定义的

concurrent(P, A, B, Q) = aux(P, Q)

这里 aux(P, Q)=

λx.if="BLEEP or Q="BLEEP then "BLEEP

else if ismember(x,A) ∧ ismember(x,B) then aux(P(x)

Q(x))

else if ismember(x,A) then aux(P(x)，Q)

else if ismember(x,B) then aux(P，Q(x))

else"BLEEP

2.3.3 迹(Traces)

假设(P ‖ Q)的迹为 t。则 t 中属于 αP 的每个事件都是在 P 的生存期内发生的事件；不属于 αP 的事件发生时，不需 P 参加。(t↾αP)是有 P 参加的所有事件组成的迹，是 P 的一个迹。同理，(t↾αQ)也是 Q 的一个迹。另外，t 中每个事件如不在 αP 中就一定在 αQ 中。由此，我们有 L1

L1 traces(P ‖ Q) = { t | (t↾αP) ∈ traces(P)

∧ (t↾αQ) ∈ traces(Q)

∧ t ∈ (αP ∪ αQ)* }[130]

下一个规则是关于算子/s 对并行组合有分配律，即

L2 (P ‖ Q)/s = (P/(s↾αP)) ‖ (Q/(s↾αP)

当 αP=αQ, 则有 s↾αP = s↾αP = s

在这种情况下这些法则就与 2.2.3 节中的完全一样了。

例子

X1 (参阅 2.3 节 X1)

[130] P ‖ Q 的迹一定都满足：如果过滤 P 的字母表，一定属于进程 P 的迹；如果过滤 Q 的字母表，一定属于进程 Q 的迹；属于 P 和 Q 的字母表的幂级的组合。

假设 t1 = <coin, clink, curse>

则 t1↾αNOISYVM = <coin, clink>　　属于 traces(NOISYVM)

且 t1↾αCUST = <coin, curse>　　属于 traces(CUST)

因此 t1 ∈ traces(NOISYVM‖CUST)

我们可类似地推断

<coin, curse, clink> ∈ traces(NOISYVM ‖ CUST)

这说明 curse 和 clink 的记录顺序可以不同。它们有可能还会同时发生，只不过我们已决定不提供记录两个事件同时发生的方法罢了。 □

总之，(P ‖ Q)的迹是 P 和 Q 的迹的交错混合在一起，其中 P 和 Q 字母表中共有的事件只发生一次。如果 αP ∩ αQ = {}，则 P 和 Q 的迹是单纯的互相交错(1.9.3 节)，例如 2.3 节中 X2。另一种极端的情形，即 αP=αQ，迹中的每个事件都同时属于 P 和 Q 的字母表，此时(P ‖ Q)的含义就与交互作用的含义完全相同了(2.2 节)。

L3A 如果 αP ∩ αQ = {}

traces(P ‖ Q) = { s | ∃ t: traces(P); ∃ u: traces(Q) • s interleaves(t, u)}

L3B 如果 αP = αQ

traces(P ‖ Q) = traces(P) ∩ traces(Q)[131]

2.4 示意图(Pictures)

我们用一个标有 P 的方框来表示字母表为{a, b, c}的进程 P，由这个方框再引出几条线段，每条线段都用字母表中不同事件表明(图 2.1)。类似地，字母表为{b, c, d}的进程 Q 如图 2.2 所示。

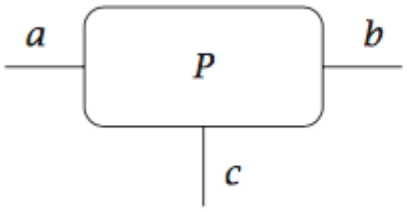


图 2.1

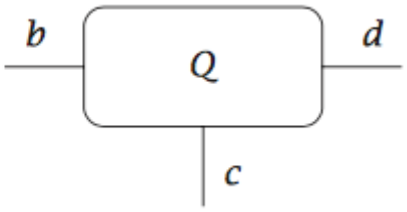


图 2.2

[131] 如果没有单独只属于某个进程的事件，则 P ‖ Q 的迹只能是两个进程迹的交集(含空)。

当把这两个进程连到一起并运行时，所得的系统的示意图就象是个网络，具有相同标记的线段连接在一起，但是如果线段上标记的事件只属于其中一个进程的字母表，则这条线段不与任何其它线段连接(图 2.3)。

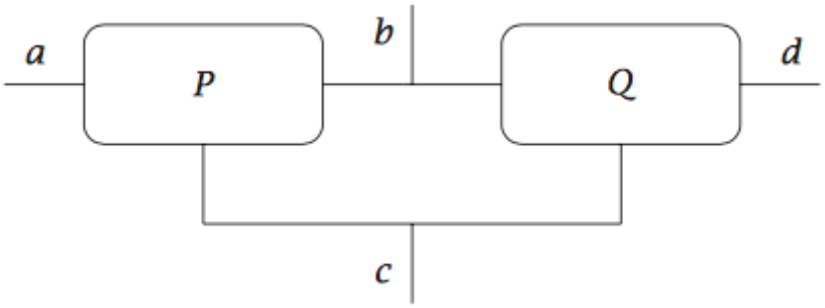


图 2.3

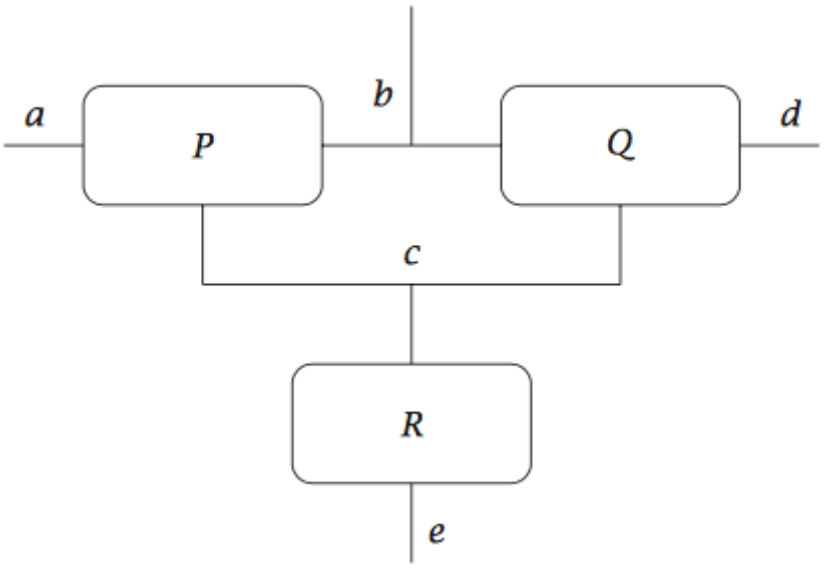


图 2.4

在 2.3 的基础上也可以再添加进程 R，αR = {c, e}，进程 R 加到示意图上去，形成图 2.4。在这个框里，事件 c 的发生需要三个进程 P, Q 和 R 同时参加，事件 b 需要 P 和 Q 两个进程同时参加，其它的事件如 a, d, e 就只与单个进程有关了。这类的示意图被称做为连接框图。

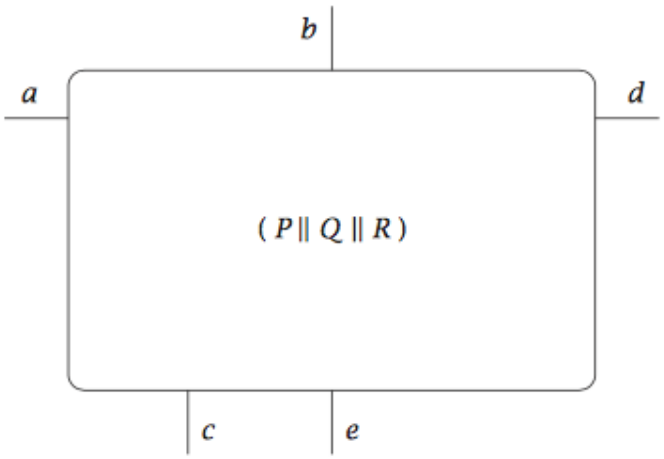


图 2.5

但这些示意图很容易引起误解。其实，由这三个进程构成的一个系统也还是一个进程，因此应当画成一个方框(如图 2.5)。例如，整数 60 可由三个数乘积(3×4×5)构成；但一旦构成后，它也只是一个整数而已，和它的构成方式就不再有什么关系了，甚至也不可能从它本身观察到它的构成方式了。

2.5 例子：哲学家就餐问题(The Dining Philosophers)

古时候，一位非常富有的慈善家出钱资助一所学院，让学院为五位杰出的哲学家提供食宿。每个哲学家都有一个房间，他可以在里面专心思考；但是餐厅是公共的，里面摆了一张圆桌、五把椅子，每把椅子上都标有一位哲学家的名字。他们的名字是 $PHIL_0$, $PHIL_1$, $PHIL_2$, $PHIL_3$, $PHIL_4$。并且以逆时针次序围坐在桌旁。在每位哲学家的左侧放着一把金叉，桌子正中间摆着一个大碗，里面盛着意大利细条通心面，而且随时地有人添加着。

哲学家本来就是要把他的主要时间用于思考的；但如果觉得饿了，他就去餐厅，坐在他的椅子上，拿起放在左边的叉子，吃一顿细条通心面。可事情并不这么简单，要把缠结的通心面送到嘴里，必须用第二把叉子。所以这位哲学家非得把他右边的叉子也拿起来不可。一旦吃完了，他就放下两把叉子，站起来回到他房间里继续思考去。当然，一把叉子一次只能由一个人用。如果有另外一位哲学家要用这把叉子，他只能等头一位哲学家用完了再说

2.5.1 字母表(Alphabets)

现在我们为这个系统构建一个数学模型。首先我们必须选出相关事件的集合。对 $PHIL_i$，集合定义为

$\alpha PHIL_i$ = {i.sits down, i.gets up,

i.picks up fork.i, i.picks up fork.(i⊕1),

i.puts down fork.i, i.puts down fork. (i⊕1) }

这里⊕是模数为 5 的加法，因此 i⊕1 就是第 i 位哲学家右侧的邻座编号。

注意每个哲学家的字母表是相互独立的，他们不在一块做任何事件，所以就不相互作用或通信——这是那个时代哲学家们的行为的客观反映。

在我们这个小剧目里的其它角色就是那五把叉子，每把叉子上贴的号与它主人的号相同。一把叉子或由其主人使用，或由叉子另一边的，主人的邻居使用。则第 i 把叉子的字母表为

$\alpha FORK_i$ = {i.picks up fork.i, (i ⊖ 1).picks up fork.i,

i.puts down fork.i, (i ⊖ 1).puts down fork.i}

这里⊖表示模数为 5 的减法。

这样，除了坐下和站起来的事件之外，其它的每个事件都需要两个密切相关的演员参加，一位哲学家和一把叉子，如连接图 2.6 所示。

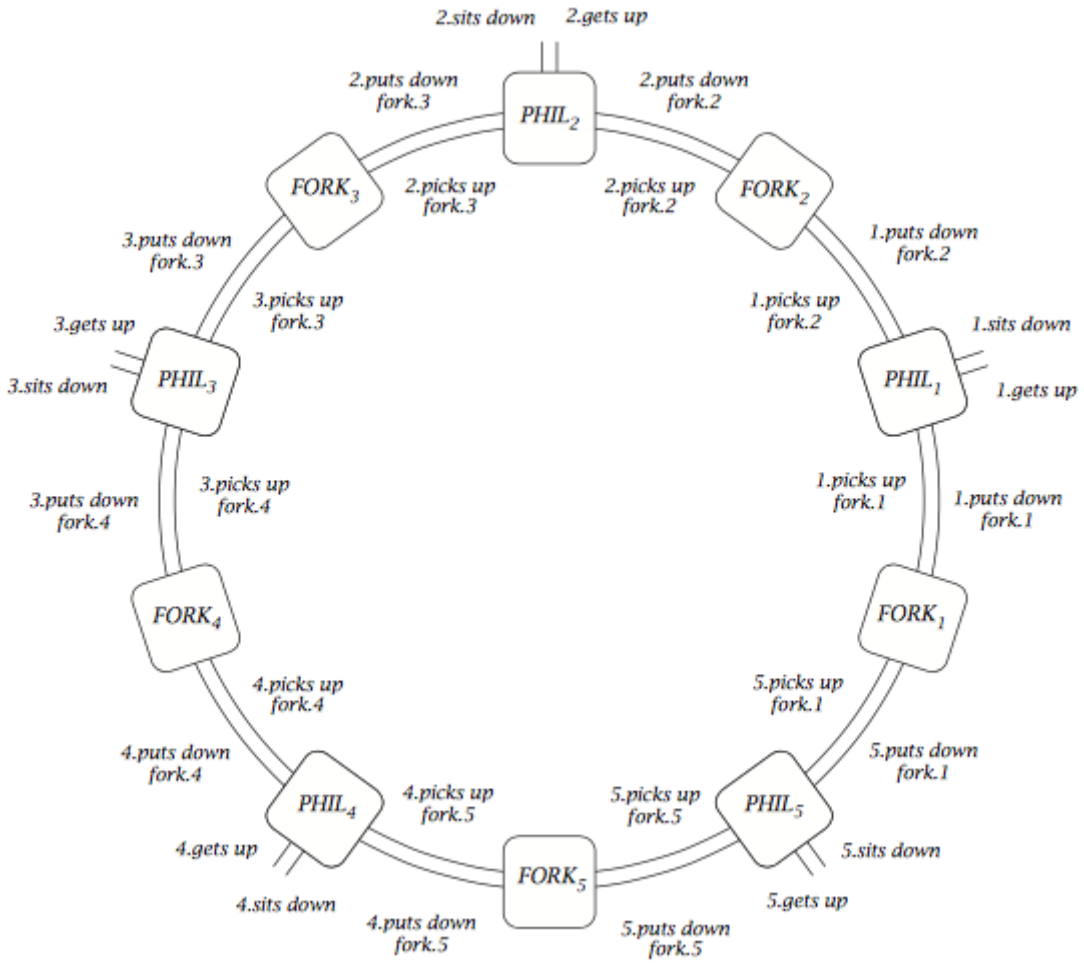


图 2.6

2.5.2 行为(Behaviour)

除了我们有意忽略的思考和吃饭两件事件外，哲学家的生活就是六件事件为一个周期，不断重复，即

$PHIL_i$ = (i.sits down →
i.picks up fork.i →[132]
i.picks up fork.(i ⊕ 1) →
i.puts down fork.i →
i.puts down fork.(i ⊕ 1) →
i.gets up → $PHIL_i$)

[132] 拿左手的叉子。i.picks up fork.(i⊕1)拿右手的叉子。

叉子扮演的角色很简单；就是被坐在旁边的哲学家重复地拿起来或放下(两个动作都由同一个人做)，即

$FORK_i = (i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow FORK_i \mid (i \ominus 1).picks\ up\ fork.i \rightarrow (i \ominus 1).puts\ down\ fork.i \rightarrow FORK_i)$

所以整个学院的行为就是各个组成部分的行为的并发组合

$PHILOS = (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4)$

$FORKS = (FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4)$

$COLLEGE = PHILOS \parallel FORKS$

这个故事稍加有趣变动，如果允许哲学家以任意左右次序拿起或放下他们的叉子[133]，即分别考虑哲学家每只手的行为，每只手都能拿起放下旁边的那把叉子，但是要想坐下或站起来，必须两只手帮忙。这样就有

$\alpha LEFT_i = \{i.picks\ up\ fork.i, i.puts\ down\ fork.i,$
$i.sits\ down, i.gets\ up$

$\alpha RIGHT_i = \{i.picks\ up\ fork.(i \oplus 1),$
$i.puts\ down\ fork.(i \oplus 1), i.sits\ down, i.gets\ up\}$

$LEFT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.i \rightarrow$
$i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i)$

$RIGHT_i = (i.sits\ down \rightarrow i.picks\ up\ fork.(i \oplus 1) \rightarrow$
$i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i)$

$PHIL_i = LEFT_i \parallel RIGHT_i$

坐下和站起来是需要 $LEFT_i$ 和 $RIGHT_i$ 的同步工作[134]，这一点就保证了只有当哲学家入座后，才能拿起叉子来。除此之外，对两把叉子的操作则是任意交叉的。

我们把这个故事再变一变，当哲学家坐下后，他可以把叉子拿起来或放下去很多次。这样，他两只手的行为就要有所改动，允许有一种重复的动作，例如

$LEFT_i = (i.sits\ down \rightarrow$
$\mu X . (i.picks\ up\ fork.in \rightarrow i.puts\ down\ fork.i \rightarrow X$
$\mid i.gets\ up \rightarrow LEFT_i))$

---

[133] 之前的行为定义是：先拿自己的叉子(左手)，然后拿别人的(右手)。

[134] $PHIL_i = LEFT_i \parallel RIGHT_i = (i.sits\ down) \rightarrow ((i.picks\ up\ fork.i \rightarrow i.puts\ down\ fork.i \rightarrow i.gets\ up \rightarrow LEFT_i) \parallel (i.picks\ up\ fork.(i \oplus 1) \rightarrow i.puts\ down\ fork.(i \oplus 1) \rightarrow i.gets\ up \rightarrow RIGHT_i))$。参阅 2.3.1 的 L6， 显然 $PHIL_i$ 的行为会是坐下来之后，左手，右手的 4 个独立事件的各种组合。然后在都放下叉子后，同步在站起来的事件上。然后递归回去。

2.5.3 死锁(Deadlock)!

当数学模型建立起来后，揭示了一个严重的问题。如果说这五位哲学家在大致同一时间里都饿了，他们都到餐厅里坐下来，拿起属于他的那把叉子，然后去拿另一把—但这另一把叉子却不在了。在这种尴尬的情况下，他们必定要挨饿。尽管每个角色都有能力做更多的动作，但他们中的任意两个都不能合作地往下发展[135]。

尽管如此，我们这个故事的结尾不会这么惨，一旦发现问题，总可以找到解决的办法。譬如说，他们中的一位拿第一把叉子时总是先拿右邻的—但很难使他们一致同意由谁这样做！同样的理由，也难决定再买一个叉子专归某一位，而再买五把又太贵了。最后采取的解决办法是找位男仆，他的职责就是为每个人挪椅子，他的字母表是

$\cap_{i=0}^{4}$ {i.sits down, i.gets up}

这位仆人得到秘密指示，最多只能让四个人同时吃饭。则他的行为如果用联立递归式定义就非常简单。

U = $\cap_{i=0}^{4}$ {i.gets up} D = $\cap_{i=0}^{4}$ {i.sits down}

$FOOT_j$定义仆人帮助 j 位哲学家就座后的行为

$FOOT_0 = (x: D \rightarrow FOOT_1)$

$FOOT_j = (x: D \rightarrow FOOT_{j+1} \mid y: U \rightarrow FOOT_{j-1}) \quad j \in \{1, 2, 3\}$

$FOOT_4 = (y: U \rightarrow FOOT_3)$

没有死锁问题的学院定义为

$NEWCOLLEGE = (COLLEGE \parallel FOOT_0)$[136]

这个哲学家就餐的故事是来自 Edsger W. Dijkstra。雇一位仆人的主意则来自 Carel S. Scholten。

2.5.4 死锁不存在的证明(Proof of absence of deadlock)

在 COLLEGE 进程中，死锁问题并非显而易见；因此，如果说在 NEWCOLLEGE 中就没有死锁问题，需要认真地证明才行。我们必须要证明的东西可形式地叙述为

[135] 可能更是一个 starving 的状态。每个哲学家都在一直试图反复的看是否可以拿到右手的叉子。

[136] 一旦餐桌有了 4 个哲学家在就餐，FOOT 进程会进入只响应 i.gets up 的事件。从而导致不会出现系统的迹还出现 5 个哲学家争抢的现象。

(NEWCOLLEGE / s) ≠ STOP　对全部 s ∈ traces(NEWCOLLEGE)都成立。

证明时先任取这个进程的一个迹 s，然后证明在任何情况下，至少能找到一个事件，以延伸 s 后得到的序列扔在 traces(NEWCOLLEGE)。我们定义入座的哲学家的人数为

seated(s) = #(s ↾ D) - #(s ↾ U)　　其中 U 和 D 的定义如上。

因为(由 2.3.3 节的 L1) s ↾ (U ∪ D) ∈ traces($FOOT_0$)，我们知道有 seated(s)≤4。如果 seated(s)≤3，则至少还能再容下一个人入座，所以不会出现死锁。当 seated(s)=4 时，数一下正在吃的哲学家有几位(正吃的人都举着两把叉子)。如果有人正在吃，则正吃的一位总是可以放下他左手拿的叉子。如果没人在吃，就看着被拿在手里的叉子的个数。如果只有或还不足三把叉子被拿在手里，则有一位就座的哲学家还可以拿起他左边的叉子。如果已拿起四把，则坐在空位左侧的那位不仅左手握叉，而且还能把他右边的叉子也拿起来。如果五把叉子都被拿起来了，则这四位中至少有一位一定在吃了。

非形式化的描述这个具体例子的行为的证明分析要分很多种情况[137]。现在我们换一种证明方法：设计一个计算机程序，用来研究这个系统所有可能的行为，来寻找死锁[138]。一般说来，这样的一个程序无法保证死锁的不存在，因为我们永远无法知道这个程序是否已经遍历了全部可能行为[139]。但是象在 COLLEGE 这样的有限状态系统中，所需考虑的迹，只是有穷多个，因为它们的长度不会超过可由状态数计算出的某个已知上限。(P ‖ Q)的状态数不超过 P 的状态数与 Q 的状态数之积。由于每个哲学家有六个状态，每把叉子有三个状态，则 COLLEGE 的状态总数不超过

$6^5 \times 3^5$　　大约 1.8×10^6

由于仆人(Footman)的字母表已经包含在 COLLEGE 中，所以 NEWCOLLEGE 的状态不会比 COLLEGE 多。又因为在每个状态的中几乎都包含了两个或更多的可能事件，我们要考虑的迹的个数就超过 $2^{1.8\times10}$ 了。尽管是有穷多个，但要让一个计算机程序遍历全部这些迹，简直是不可能的。因此，即使对一个很简单的有限状态进程来说，其死锁的不存在性的证明还是很艰难，必须留给并发系统的设计人员去处理。

[137] 类似穷举所以可能的后续行为。
[138] 有兴趣的读者可以参阅 Lesie Lamport 的 TLA+。
[139] 或者是可以遍历，但状态空间爆炸，遍历需要指数级别的时间。

2.5.5 无限抢先(Infinite overtaking)[140]

除了死锁之外，吃饭的哲学家还面临着另一个危险——总是被他的邻位抢了先，假设一位入座的哲学家左手不太灵活，又有个非常贪吃的左邻座。还没等他把左边的叉子拿起来，那位邻座就闯进来，坐下来，一下子就把他左右的两把叉子都拿起来了，半天吃不完。好不容易吃完了放下两把叉子，离开座位。可是马上又饿了，又跑回来，坐下，抓起两把叉子又饱餐一顿，只苦了等了好久的那位右边的哲学家，他来不及拿到那把他们两个人合用的叉子。这个周期可能无限循环下去，这样就会有一位入座的哲学家总也吃不成。

这个问题没有办法解决，因为如果真有位那么贪吃的哲学家则肯定有人(不是他就是他的邻座)要长时间挨饿。实在没有什么聪明的办法使大家都满意，只能再多买几把叉子，多添几次通心面，才可以解决这个问题。

但是，如果保证入座的人最终都能吃到，可以修改一下男仆的行为：在他帮着一位哲学家入座后，他就一直等到这个人把两把叉子都拿起来之后，才去帮着这个人的左右邻居入座。

2.6 符号变换(Change of symbol)

在前一节所举的例子中有两组进程，一组是哲学家，一组是叉子；每一组中进程的行为相似，只是执行的事件的名字不同。这一节里，我们介绍一个很方便的方法，用以定义行为相似的成组进程。设 f 为一个单射函数，它将 F 的字母表映射到符号集合 A 上去

f ： $\alpha P \rightarrow a$

我们定义 $f(P)$为这样的一个进程：当 P 执行事件 c 时，$f(P)$就执行事件 $f(c)$。它遵从以下规则

$\alpha f(P)=f(\alpha P)$

$traces(f(P)) = \{f^*(s) \mid s \in traces(P)\}$

(f^*的定义看 1.9.1 节)。

[140]操作系统中的 starving 的概念。

例子

X1 过了几年之后，所有的东西都要涨价。要把通货膨胀表示出来，我们以下列方程式定义函数

f(in2p) = in10p　　f(large) = large

f(inlp) = in5p　　f(small) = small

f (outlp) = out5

则新的自动售货机就是

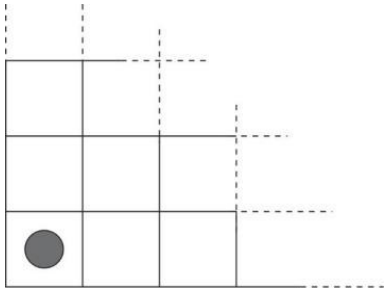
NEWVMC = f(VMC)　　□

X2 一个筹码，行为跟 CT_0 相同(1.4.4 节 X2)，不同的是，它是向左和向右移动，而不是向上或向下移动，令

f(up) = right, f(down) = left, f(around) = around, $LR_0 = f(CT_0)$　　□

我们对进程的事件名做这种变换，主要的原因是使它们能在并发组合时充分发挥作用。

X3 一个筹码可在板中、上、左、右移动，这块板是上方和右方无穷的，其边界在左边沿和下边沿。



如图筹码从左下角出发，也仅在这个方格里，它才允许转圈。用第 2.3 节 X2 中的办法，将垂直和水平移动分别看作是不同进程的独立动作；但是 around 就需要两个进程同时参加，故 LRUD = $LR_0 \| CT0$。

X4 我们希望把两个 COPYBIT(参看 1.1.3 节 X7)连接起来，这样由第一个 COPYBIT 输出的每个数位同时成为第二个 COPYBIT 的输入。首先，我们改变一下用作内部通信的事件名；引入两个新的事件 mid.0 和 mid.1，再定义函数 f 和 g，以变换一个 COPYBIT 的输出及另一个 COPYBIT 的输入，即

f(out.0) = g(in.0) = mid.0

f(out. 1) = g(in.1) = mid.1

f(in.0) = in.0, f (in.1) = in.1

g(out.0) = out.0, g(out.1) = out.1

我们需要的结果就是

CHAIN2 = f(COPYBIT) ‖ g(COPYBIT)[141]

值得注意的是，由 *f* 和 g 的定义可见，‖左边的运算对象的输出 0 或 1 与其右边运算对象的输入 0 或 1 完全是同一事件(mid.0,或 mid.1)。这样就给出了图 2.7 所示的二进制数字的同步通信的模型，该通信是在一条连接了两个运算对象的通道上进行的。

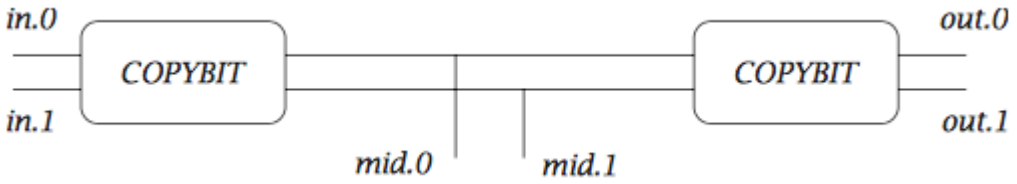


图 2.7

左边的运算对象决定在连接通道上传送哪一个值，而右边的运算对象则准备好或者执行事件 Mid.0，或者执行事件 Mid.1。所以确定发生哪个事件的总是输出进程。并发进程之间的这种通信方法会在第四章中推广到其通用的情形。

注意内部通信 Mid.0 和 Mid.1 包含在组合进程的字母表中，并且可被进程的环境观察到(甚至可受环境控制)。有时我们希望把这类内部事件忽略或屏蔽掉，但在一般情况下，做这类屏蔽可能会引入非确定性，所以我们把这个问题退后到 3.5 节去处理。

□

X5 我们想把计算机程序中使用的布尔变量的行为表示出来。字母表中的事件有

assign0　给变量赋 0 值

assign1　给变量赋 1 值

fetch0　当变量为 0 时取出变量的值

fetch1　当变量为 1 时取出变量的值

变量的行为与饮料机(1.1.4 节 X1)的行为非常相似，所以我们定义

BOOL = *f*(DD)

[141] 在 CHAIN2 进程里，通过函数 f 和 g 的变换后，符号表或者说能触发的事件是：作用在第一个 COPYBIT 上的 in.0，in.1，作用在第二个 COPYBIT 上的 out.0，out.1。中间的 mid.0 和 mid.1 事件。值得非常注意的是：在 CSP 中，只要是出现在符号表中的事件，都是可以被外界所观察到的(Observer)。这也是为什么作用在两个进程上的并发算子得到的迹一定是两个进程的共同的行为的凝聚。否则，观察者会记录不下来复合进程的迹。

函数 f 的定义留作小小的练习。要注意的是这个布尔变量在未被赋值前是不会给出它的值的。要取一个未被赋值的变量的值，将导致死锁—这恐怕是程序错误中最容易发觉的出错方式了，因为最简单的分析复盘就能将这类错误指出。 □

把函数 f 作用于 P 的树形图各枝的标记上，就得到了 f(P)的树形图。由于 f 是 1 对 1 的函数映射，故该变换仍能保持树的原结构，而且由同一结点引出的分叉上的标记仍互不相同。例如 NEWVMC 的示意图如图 2.8。

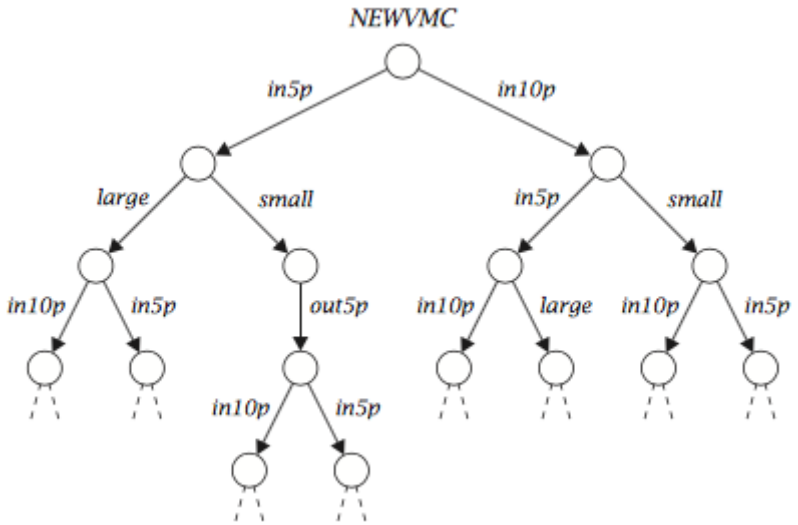


图 2.8

2.6.1 法则(Laws)

在应用一对一的函数对进程进行变换符号时不会改变进程的行为结构。这一点由以下事实得出：1-1 函数运算对其它一切算子均满足分配律。具体可详见下面的法则。首先我们要用到几个辅助定义如下：

f(B) = $\{f(x) \mid x \in B\}$

f^{-1}　　f 的逆函数

$f \circ g$　f 和 g 的复合

f^*　　定义见 1.9.1 节

强调 f 是单射函数，主要是因为在法则中要用到 f^{-1}。

经过符号变换后，STOP 在改变后的字母表中仍什么都不做，即

L1 $f(STOP_A) = STOP_{f(A}$

在选择的情况下，供选择的符号变了，且进程的后续行为也要相应变化，故

L2 $f(x : b \rightarrow P(x)) = (y : f(B) \rightarrow f(P(f^{-1}(y))))$

等式右边的 f^{-1} 可能需要解释一下。我们说过 P 为一函数，它要根据由集合 B 中选出的某事件 x 得到一个进程，但等式右边的变量 y 是集合 $f(B)$的元素。对 P 来说，y 的对应事件就是 $f^{-1}(y)$，且 $f^{-1}(y)$在 B 中(因为 $y \in f(B)$)。执行完 $f^{-1}(y)$之后，P 的行为就是 $P(f^{-1}(y))$，而且进程 $P(f^{-1}(y))$的动作在 f 的作用下还要继续变换。

符号变换显然对并行组合有分配律，故

L3 $f(P \parallel Q) = f(P) \parallel f(Q)$

它对递归式的分配律更稍微复杂些，且使进程字母表相应地变化，有

L4 $f(\mu X: A \bullet F(X)) = (\mu Y : f(A) \bullet f(F(f^{-1}(y))))$

等右边的 f^{-1} 又有点让人迷惑不解。以前讲过，要使等式左边的递归式有效，要求函数 F 的自变量为字母表为 A 的进程，而且函数 F 的计算结果也是个进程，它与自变量的字母表相同。L4 中等式右边，变量 Y 的变化范围是所有字母表为 $f(A)$的进程，因此只有当其字母表经反变回到 A 之后，才能做为 $F(f^{-1}(Y))$的字母表为 A，应用 f 就将字母表变成 $f(A)$，以此保证法则 L4 等式右边递归式的有效性。

两次符号变换的复合就定义为两个符号变换函数的复合，即

L5 $f(g(P)) = (f \circ g)(P)$

经变换符号后的进程的迹，是将原进程的迹中的各个符号都变换后得到的，故

L6 $\text{traces}(f(P)) = \{f^*(s) \mid s \in \text{traces}(P)\}$

最后一条法则的解释说明与 L6 类似

L7 $f(P) / f^*(s) = f(P/s)$

2.6.2 进程标记(Process labelling)

符号变换在组建由类似的进程组成进程群体时特别有用，这些类似进程并发操作，对其外部环境提供相同的服务，但是相互之间不发生联系。这意味着它们的字母表必须互不相同而且互不相交。为了做到这一点，我们把每个进程都标记上不同的名字；再把进程的每个事件也标记上与其所属进程相同的名字。做了标记的事件是个二元组 l.x，其中 l 是标记，x 是代表该事件的符号。

标记为 l 的进程 P 表示为

l : P

每当 P 要执行 x 时，l：P 就执行 l.x。定义 l：P 所用的函数是 f_l

$f_l(x) = l.x \qquad x \in \alpha P$

而标记的定义是

$l: P = f_l(P)$

例子

X1 两台并排放置的自动售货机记为

(left : VMS) ‖ (right : VMS)

这两个进程的字母表是不相交的，而且在哪台机器上发生的事件就用那台机器的名字做上标记。万一在并排放置之前没给他们赋名，则每个事件的发生都要求它们同时参与。这样一来这两台机器就跟一台没什么区别了。由这一事实我们推出

(VMS‖VMS) = VMS □

有了进程标记，就可以象使用高级程序设计语言里的变量那样使用进程了，这类变量只在具体用到它们的程序块中局部地说明[142]。

X2 布尔变量的行为的模型是 BOOL(2.6 节 X5)。有一个程序块其行为表示为进程 USER。该进程对两个布尔变量 b 和 c 进行赋值和取值。这样 αUSER 中就包括了如下的复合事件

b.assign.0　　给 b 赋 0 值

c.fetch.1　　当 c 的值为 1 时取出其当前值

进程 USER 与它的两个布尔变量并行运行

b : BOOL‖ c : BOOL ‖ USER

在程序 USER 内部，可以等效地表示

b:=false; P　为　　(b.assign.0 → P)

b :=¬ c; P　为　　(c.fetch.0 → b.assign.l→ P

|c.fetch.l → b.assign.0 → P)

要注意的是，我们是通过允许变量在 fetch0 和 fetch1 之间做出选择的方法，以发现变量当前值的；还要注意该选择并以适当方式影响 USER 的后续行为。 □

在 X2 及以下几例中，如果先定义了赋值，可能就更方便了，例如先定义赋值

b := false

而不是直接定义命令对

[142] 通过使用标记，可以实现对个进程行为的实例，达到变量的效果。

b := false; P

在命令对中明确提到了程序的剩余部分 P。单独定义赋值命令的方法将在第五章中介绍。

X3 进程 USER 需要两个计数变量 l 和 m。给它们赋的初值分别为 0 和 3。USER 使每个变量增值的事件为 l.down 和 m.down。检验变量是否为 0 的事件为 l.around 和 m.around。所以就可以使用进程 CT(1.1.4 节 X2)，但需用 l 和 m 作为适当标记。既有

$(l : CT_0 \| m: CT_3 \| USER)$

在进程 USER 内，可以等效地表示习惯使用的

(m := m + 1; P) 为　　$(m.up \rightarrow P)$

if l = 0 then P else Q 为　　$(l.around \rightarrow P \mid l.down \rightarrow l.up \rightarrow Q)$

注意检验变量是否为零是这样进行的：进程通过事件 l.around 检验 l 是否为零的同时，也通过事件 l.down 去将计数变量 l 减少 1。这样 l 就要在这两个事件中选择：如果 l 的值为 0，它就选择 l.around，如果它不是 0，就选择 l.down。但在后一种情况里选择了 l.down 之后，l 的值已经被减少 1，故需用事件 l.up 立即将其恢复为原值。在下一例中，恢复原值的过程要更繁琐些。

(m := m+l; P)　　由 ADD 实现

其中 ADD 递归地定义为

$ADD = DOWN_0$

$DOWN_i = (l.down \rightarrow DOWN_{i+1} \mid l.around \rightarrow UP_i)$

且 $UP_0 = P$

$UP_{i+1} = l.up \rightarrow m.up \rightarrow UP_i$

进程组 $DOWN_i$ 把 l 逐步减到 0 从而发现 l 的初值，然后由进程组 UP_i 再把这个初值分别加到 m 和 l 上，这样既恢复了 l 的初值，而且还把该值加到了 m 上。

用一组并发的进程可以等效实现一个数组变量，这些并发进程用它们在数组的下标做为标记。

X4 进程 EL 的作用是记录时间 in 是否已经发生了。第一次发生时，EL 的回答是 no，以后发生时，EL 的回答是 yes。令

aEL = {in, no, yes}

$EL = in \rightarrow no \rightarrow \mu X \bullet (in \rightarrow yes \rightarrow X)$

这个进程可用在数组中模拟小整数的集合行为

SET_3 = (O: EL) ‖ (1 : EL) ‖ (2 : EL) ‖ (3 : EL)

整个数组在试用前还可再加一次标记，得到

m : SET_3 ‖ USER

在 a(m:SET_3)中的每个事件是一个三元组，例如 m，2.in。在 USER 进程内，由

m.2.in → (m.2.yes → P |m.2.no → Q)

可取得以下效果

if 2 ∈ m then P else (m := m ∪ {2} ; Q) □

2.6.3 实现(Implementation)

要实现通用的符号变换，我们需要知道符号变换函数 *f* 的反函数 g。我们还需要确保 g 的自变量超出了 *f* 的值域时，g 会给出回答"BLEEP。符号变换的实现以 2.6.1 节 L4 为依据。令

```
change(g, P) = λx • if g(x) = "BLEEP then
        "BLEEP
        else if P(g(x)) = "BLEEP then
                "BLEEP
        else
                change(g, P(g(x)))
```

进程标记，作为一种特殊情况，它的实现就更简单了。我们用原子对 cons("l, "x)。代表复合事件 l.x。(l : p)的实现可以表示如下

```
label(l, P) = λ y • if null(y) or atom(y) then[143]
        "BLEEP
    else if car (y) ≠ I then
        "BLEEP
         else if P(cdr(y)) = "BLEEP then
                 "BLEEP
                        else
                        label(l, P(cdr(y)))
```

[143] 在 1985 年版的 CSP 书里，用的是 LISP 的逻辑符号“∨”，而非 Hoare 在 2015 年电子版用的 "or"。

2.6.4 多重标记(Multiple labelling)

标记的定义可以扩展到允许事件在集合 L 中任取标记 l。设 P 是一进程，定义(L:P)是行为与 P 完全相同的进程，只是一旦 P 执行 c 时，(L:P)即执行事件 l.c($l \in L, c \in \alpha P$)。标记 l 的每次选择均由(L:P)的环境独立完成。

例子

X1 一个年轻的男仆 LACKEY，这个男仆服侍他的主人入座、离座，主人吃饭时他要站在身后，于是有

aLACKEY = {sits down, gets up}

LACKEY = (sits down→gets up→ LACKEY)

要教会 LACKEY 为五个主人服务(但每次只为其中一个服务)。我们定义

L = {0,1,2,3,4}

SHARED LACKEY = (L: LACKEY)

因此，当那位男仆(2.5.3 节)去度假时，这个共享的男仆可以被雇来为就餐的哲学家服务，不至于出现死锁。当然，在这段时间里，哲学家们更要挨饿了，因为每次他们中只能有一个人可以上桌就餐。

如果 L 中的标记不止一个，L：P 的树形图与 P 的树形图尽管类似，但从一个节点引出的分支增多了，从这一意义上说，L:P 的树要更茂盛些。如 LACKEY 的示意图是一根光秃秃的树干没有树杈(图 2.9)。

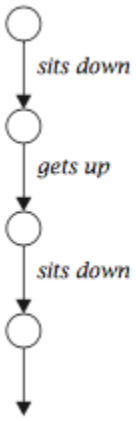


图 2.9

而{0,1}：LACKEY 的示意图则是一个二叉树(图 2.10) SHARED LACKEY 的示意图就更复杂了。

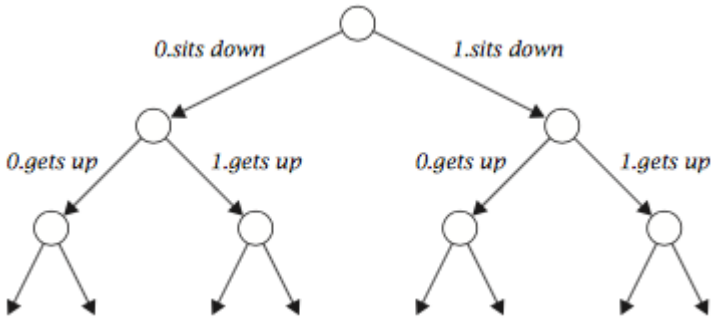


图 2.10

总之，有了多重标记，假设预先知道这些标记的符号，就能使多个做了标记的进程共享某一个进程的服务。共享技术将在第六章中更充分地加以讨论。

2.7 功能描述(Specifications)

假设 P 和 Q 为并发运行的进程，并已证明 P sat S(tr)和 Q sat T(tr)

设 tr 为(P ‖ Q)的一个迹。由 2.3.3 节 L1 可知(tr ⌈αP)是 P 的迹，因而它也满足 S，即

S(tr ⌈ αP)[144]

同理，(tr ⌈ αQ)为 Q 的迹。即

T(tr ⌈ αQ)

上面的论证对(P ‖ Q)的每个迹都成立。于是我们可以推出，

(P ‖ Q) sat (S(tr⌈αP) ∧ T(tr⌈αQ))[145]

可以通过非正式的归纳成如下法则

L1 如果 P sat S(tr)

Q sat T(tr)

则(P ‖ Q) sat (S(tr⌈αP) ∧ T(tr⌈αQ))

例子

X1 (参见 2.3.1 节 X1)

设 αP = {a, c}　αQ = {b, c}

P = (a → c → P)

Q = (c → b → Q)

[144] S(tr ⌈ αP)的输入(tr ⌈ αP)是 P 的迹。因此，满足 S。

[145] 同时满足两个进程 P 和 Q 的迹，就是并发进程 P ‖ Q 的迹。

我们要证

(P ‖ Q) sat 0 ≤ tr ↓ a - tr↓b ≤ 2

对 1.10.2 节中 X1 的证明加以修改，容易证得

P sat (0 ≤ tr↓a - tr↓c ≤ 1)

Q sat (0 ≤tr↓c - tr↓b ≤ 1)

由 L1 就有

(P ‖ Q)

sat (0 ≤ (tr⌈αP)↓a - (tr⌈αP)↓c ≤ 1 ∧

0 ≤ (tr ⌈ αQ)↓c - (tr⌈αQ)↓b ≤ 1)

⇒ 0 ≤ tr↓a - tr↓b ≤ 2 [当 a∈A，有(tr⌈A)↓a = tr↓a] □

由 sat 的法则我们知道 STOP 满足一切能够被满足的描述，因此基于 sat 法则的推理无法证明一个进程死锁的不存在性[146]。在 3.7 节中我们将给出更强大的法则来。而在目前，要消灭死锁的危险，一个办法是如 2.5.4 节中我们所做的那样，逐步仔细的证明进程的无死锁，另一个办法是如 2.3.1 节中 X1 中我们所做的那样，证明用并行组合算子定义的进程与没有用这个组合算子定义的非停止进程等价。可是这类证明总要牵扯到一大串冗长乏味的代数变换。因此只要可能，我们还是应该用通用法则来证明，例如

L2 设 P 和 Q 永不停止，且(αP ∩ αQ)最多包含一个事件，则(P ‖ Q)也永不停止[147]。

例子

X2 在 X1 中定义的进程(P ‖ Q)永不停止，因为

αP ∩ αQ ={c}

符号变换的证明规则是

L3 如果 P sat S(tr)

则 f(P) sat S(f^{-1*}(tr))

法则结论中的 f^{-1*}可能要解释一下。设 tr 为 f(P)的迹，则 f^{-1*}(tr)为 P 的迹。L3 中的前提说明 P 的每个迹都满足 S，由此可得 f^{-1*}(tr)满足 S，这正是 L3 的结论。

[146] 死锁在 CSP 里的表现为 STOP 行为。

[147] 如果超过两个共同的事件，就开始涉及同步的问题。如果没有聚合出来共同的迹或者说，没有相同的可被记录的行为，复合的并发进程就会无所适从，陷入死锁的处境。而只有一个共同的事件不会出现无所适从的问题。因为，这个事件后续的事件可以是属于任何一个进程的，不相关的事件，不需要同步的控制。

2.8 确定性进程的数学理论(Mathematical theory)

在描述刻画进程的过程中，我们陈述了很多法则，并且有时还将它们用于以些命题的证明。如果说这些法则是经过证明了的话，证明的过程也仅限于非形式的解释和略微说明为什么我们认为它们是成立的理由，对于一位有应用数学或工程师的直觉的读者来说，那些解释也就够了，但还免不了要提些问题，譬如，这些法则是够真的成立？它们会不会相互矛盾？还有没有更多的法则？或者，它们是否完全，即有了这些法则就可证明有关进程的一切性质？是否能用更少的法则证同样多的事实呢？只有经过更深入的数学研究才能找到这类问题的答案。

2.8.1 基本定义(The basic definitions)

在建立一个物理系统的数学模型时，用物体的直接或间接可观察或测量的属性定义其相关的基本概念，是一种很好的方法。对确定性进程 P，它的两个比较熟悉的属性是

αP　　P 原则上能执行的事件集合

traces(P)　　P 能实际参加的所有事件序列的集合

我们已解释过它们为何一定满足 1.8.1 节中的法则 L6，L7，L8。下面我们考虑一个任意集合二元组(A, S)，也满足这三个法则。则这个集合对也就唯一决定了一个进程 P，它的迹集是 S。其构成如下

设 $P^0 = \{x | < x > \in S\}$

且对所有 $x \in P^0$，P(x)进程定义为一个迹为$\{t | <x>^\wedge t \in S\}$的进程

则有形式定义如下

$\alpha P = A$

$P = (x: P^0 \rightarrow P(x))$

另外，集合对(A,S)也可被下列方程式还原出来

$A = \alpha P$

$S = traces(x : P^0 \rightarrow P(x))$

这样在每个进程 P 和集合对(αP，traces(P))之间就有了一种一一对应的关系。在数学上，有了这种关系就足以证明这两个概念是等同的，因为可以用其中一个概念定义另一个概念。

D0 一个确定性进程就是一个集合对

(A, S)

其中 A 是符号的任意集合

而且 S 是满足以下两个条件的 A*的任意子集合

C0 <> ∈ S

C1 $\forall s, t \bullet s^\frown t \in S \Rightarrow s \in S$

满足这个定义的最简单的例子就是不动作的进程，即

D1 $STOP_A = (A, \{<>\})$

而另一个最极端的例子是任何时刻动作都可以的进程

D2 $RUN_A = (A, A^*)$

现在可以形式地定义有关进程的各种算子了，即通过说明运算结果的字母表和迹是如何由运算对象的字母表和迹演变出来的。

D3 $(x{:}b \rightarrow (A, S(x))) = (A, \{<>\} \cup \{<x>s \mid x \in B \wedge s \in S(x)\})$ 其中 $B \subseteq A$

D4 $(A, S)/s = (A, \{t \mid (s^\frown t \in S\})$ 其中 $s \in S$

D5 $\mu X: A \bullet F(X) = (A, \cup_{n \geq 0} traces(F^n(STOP_A)))$ 其中 F 为卫式

D6 $(A, S) \parallel (B, T) = (A \cup B, \{s \mid s \in (A \cup B)^* \wedge (s \restriction A) \in S \wedge (s \restriction B) \in T\})$

D7 $f(A, S) = (f(A), \{f^*(s) \mid s \in S\})$ 其中 f 为 1-1 函数

当然我们还有必要证明这些定义中的等式右侧确实是进程，也就是说证明它们满足 D0 的条件 C0 和 C1。幸运的是，要证明这些很容易。

但是 D0 还不是进程概念的完美无缺的定义，这一点在第三章中将越来越明显，因为 D0 没有把进程可能具有的非确定性体现出来。所以我们还需要一个更普遍、更复杂的进程定义。非确定性进程的一切法则对确定性进程也都成立。但确定性进程遵从一些额外的法则，例如

$P \parallel P = P$

为了避免引起混乱，在本书中我们引述的法则不仅可以万无一失地应用于确定性进程，而且完全适用于非确定性进程(除了 2.2.1 L3A, 2.23 L1, 2.3.1 L3A, 2.3.3 L1, L2, L3。这些规则在含有进程 CHAOS(3.8 节)中是不成立的。

2.8.2 不动点理论(Fixed point theory)

这一节的目的是给出对于递归基本理论证明的一个轮廓，这个基本定理是：一个递归定义的进程(2.8.1 节 D5)是相应递归方程的一个解[148]，即

$\mu X \bullet F(X) = F(\mu X.F(X))$

证明的方法遵从 Scott 的不动点理论。

首先我们需要定义描述进程之间的一种次序关系

D1 $(A, S) \subseteq (B, T) = (A = B \wedge S \subseteq T)$

两个进程如果字母表相同，而且其中之一可以做另一进程做过的每件事——也许还能做得更多——则它们可以按这种次序比较大小。这个次序关系是一种偏序关系，并满足如下关系

L1 $P \subseteq P$

L2 $P \subseteq Q \wedge Q \subseteq P \Rightarrow P = Q$

L3 $P \subseteq Q \wedge Q \subseteq R \Rightarrow P \subseteq R$

在偏序关系里，链(chain)是元素的一个无穷序列

$\{P_0, P_1, P_2, ...\}$

并满足，对一切 i

$P_i \subseteq P_{i+1}$

我们定义这样一个链的极限(最小上界)为

$\bigsqcup_{i\geq 0} P_i = (\alpha P_0, \bigcup_{i\geq 0} traces(P_i))$ [149]

以后我们只对链式进程序列使用极限算子[150]。

如果一个偏序有最小元，而且所有的链均有最小上界，我们就说这个偏序是完全的。字母表为 A 的所有进程的集合形成一个完全偏序(缩写为 c.p.o.)，因为它满足以下法则

L4 $STOP_A \subseteq P$　只要 $\alpha P = A$

L5 $P_i \subseteq \bigsqcup_{i\geq 0} P_i$

---

[148] 递归方程 x = F(x)。可参阅 1.1.2 节对递归方程的引入。在第一章，作者是假设，只要一个 CSP 进程的 prefix 是卫士的(Guarded)，则这个递归方程存在着解，即存在一个不动点，使得在那个点上，x = F(x)。我们称那个解是：μX: A • F(X)。在 2.8 节里，作者开始试图首先形式定义一个进程和变换可以通过(A, S)来定义。然后在这个基础上，利用数学的集合论，代数的手段来推演。

[149] $\bigsqcup_{i\geq 0} P_i$ 本身通过(A, S)的方式定义为一个进程。 $\bigcup_{i\geq 0}$ traces(P_i)意味着这个进程的迹是所有 Pi 迹的并集。覆盖了一切可能的行为。

[150] 即需要满足 $P_i \subseteq P_{i+1}$ 关系的进程序列。

L6 $(\forall\ i\geq 0 \bullet P_i \subseteq Q) \Rightarrow (\bigsqcup_{i\geq 0} P_i) \subseteq Q$

更进一步，μ 算子(2.8.1 节 D5)可用极限重新正式定义为

L7 $\mu X\colon\ A \bullet F(X) = \bigsqcup_{i\geq 0} F^i(STOP_A)$

由一个 c.p.o.到另一个 c.p.o.(或到其自身)的函数 F，如果对所有链的极限算子都满足分配律，即若$\{P_i \mid i\geq 0\}$是一个进程链，

$$F(\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} F(P_i)$$

则我们说它是连续的。

(因为对所有 P 和 Q 有 $P \subseteq Q \Rightarrow F(P) \subseteq F(Q)$，因此所有连续函数都是单调的，因此前一方程的等式右端也是一个升链的极限[151]。)含多个自变量的函数 G 如果对其每个自变量分别连续，则我们定义 G 也是连续的。例如

对一切 Q

$$G((\bigsqcup_{i\geq 0} P_i), Q) = \bigsqcup_{i\geq 0} G(P_i, Q)$$

对一切 Q

$$G(Q, \bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} G(Q, P_i)$$

连续函数的复合仍是连续的；并且任意个数的连续函数作用于任意多个变量的任意组合上，所构成的表达式对其每个变量仍是连续的，例如，设 G, F 和 H 为连续函数，则

G(F(X), H(X,Y))

对 X 是连续，即

$$G(F(\bigsqcup_{i\geq 0} P_i), F(\bigsqcup_{i\geq 0} P_i), Y) = \bigsqcup_{i\geq 0} G(F(P_i), H(P_i,Y))$$ 对一切 Y

D3 到 D7 中定义的所有算子(除了/)在上述意义下都是连续的

L8 $(x\colon b \rightarrow (\bigsqcup_{i\geq 0} P_i(x)) = \bigsqcup_{i\geq 0}(x\colon b \rightarrow P_i(x))$

L9 $\mu X\colon\ A \bullet F(X, \bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} \mu X\colon\ A \bullet F(X, P_i)$ 若 F 连续

L10 $(\bigsqcup_{i\geq 0} P_i) \parallel Q = Q \parallel (\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} (Q \parallel P_i)$

L11 $f(\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} f(P_i)$

因此如果 F(X)是单纯由这些算子构成的一个表达式，则它对 X 也是连续的。

[151] $\bigsqcup_{i\geq 0} F^i(STOP_A)$是一个 chain 的 limit。

现在我们可以证明基本不动点定理了

$F(\mu X: A \bullet F(X))$

$= F(\bigsqcup_{i\geq 0} F^i(STOP_A))$ 根据 μ 定义

$= \bigsqcup_{i\geq 0} F(F^i(STOP_A))$ 因为 F 是连续的

$= \bigsqcup_{i\geq 1} F^i(STOP_A)$ 根据 F^{i+1} 定义[152]

$= \bigsqcup_{i\geq 0} F^i(STOP_A)$ 因为 $STOP_A \sqsubseteq F(STOP_A)$[153]

$= \mu X: A \bullet F(X)$ D5 μ 定义

这个证明只依赖于 F 为连续函数。而 F 是卫式，只是保证方程有唯一性[154]。

2.8.3 唯一解(Unique solutions)

在这一节里我们将更形式地处理 1.1.2 节中给出的论证过程，即证明进程的卫式递归方程只有一个唯一解。证明时，我们先要弄清楚具备解的唯一性的几个更为一般的条件。为简单起见，我们只讨论单一方程的情况，这种处理方法可以很容易地推广到联立方程组。

设 P 为一进程，n 是个自然数，我们定义$(P \upharpoonright n)$仍为一个进程，在执行它的前 n 个事件时，其动作和 P 一样，然后它就停止不动；形式定义为

$(A, S) \upharpoonright n = (A, \{s \mid s \in S \wedge \#s \leq n\})$

由定义引出

L1 $P \upharpoonright 0 = STOP$

L2 $P \upharpoonright n \sqsubseteq P \upharpoonright (n+1) \sqsubseteq P$

L3 $P = \bigsqcup_{n\geq 0} P \upharpoonright n$

L4 $\bigsqcup_{n\geq 0} P_n = \bigsqcup_{n\geq 0} (P_n \upharpoonright n)$

设 F 为进程到进程的一个单调函数，如果对所有 X，均有

$F(X) \upharpoonright (n+1) = F(X \upharpoonright n) \upharpoonright (n+1)$

则说 F 是构造性的。其含义是 F(X)的前 n+1 步的行为仅由 X 的前 n 步行为决定；

因此如果 $s \neq \langle\rangle$，就有

---

[152] $\bigsqcup_{i\geq 0} F(F^i(STOP_A)) = \bigsqcup_{i\geq 0} F^{i+1}(STOP_A)$。因此按照定义，是$\bigcup_{i\geq 0}$ $traces(F^{i+1}(STOP_A))$，等于$\bigcup_{i\geq 1}$ $traces(F^i(STOP_A))$，即从 $F(STOP_A)$开始。

[153] 这一步很精妙。因为 $STOP_A \sqsubseteq F(STOP_A)$，所以，我们补齐这个 $STOP_A$的迹，不会影响$\bigsqcup_{i\geq 1} F^i(STOP_A)$，是等价的。因此，我们得到$\bigsqcup_{i\geq 1} F^i(STOP_A) = \bigsqcup_{i\geq 0} F^i(STOP_A)$。

[154] 唯一的不动点。

$s \in traces(F(X)) = s \in traces(F(X\restriction(\#s - 1)))$

前缀运算就是一种最基本的构造性函数，因为

$(c \to P)\restriction(n+1) = (c \to (P\restriction n))\restriction(n+1)$[155]

通用选择算子也是构造性的，因为有

$(x{:}b \to P(x))\restriction(n+1) = (x{:}b \to P(x)\restriction n))\restriction(n+1)$

恒等函数 I 不是构造性的，因为

$$I(c \to P)\restriction 1 = c \to STOP$$
$$\neq STOP$$
$$= I((c \to P)\restriction 0)\restriction 1$$

现在我们可以通过构造函数来证明唯一解的基本定理

L5 设 F 是一构造性函数。方程

$$X = F(X)$$

对 X 只有一个唯一解

证明　设 X 为一任意解。首先我们用归纳法证明引理

$$X\restriction n = F^n(STOP)\restriction n$$

初始情形。$X\restriction 0 = STOP = STOP\restriction 0 = F^O(STOP)\restriction 0$

归纳：

$X\restriction(n+1)$	因为 X=F(X)
$= F(X)\restriction(n+1)$	F 是构造性的
$= F(X\restriction n)\restriction(n+1)$	F 是构造性的
$= F(F^n(STOP)\restriction n)\restriction(n+1)$	假设条件[156]
$= F(F^n(STOP))\restriction(n+1)$	
$= F^{n+1}(STOP)\restriction(n+1)$	由 F^n 定义

下面我们回到主定理去

$X = \bigsqcup_{n\geq 0}(X\restriction n)$	由 L3
$= \bigsqcup_{n\geq 0} F^n(STOP)\restriction n$	由引理
$= \bigsqcup_{n\geq 0} F^n(STOP)$	由 L4
$= \mu X{\cdot}F(X)$	由 2.8.2 节 L7

[155] c → (P⌈n) 中 c 事件是第 1 步。因此我们只要再取 n 步，就构成了 n+1 步。

[156] 自然归纳法，假设为 n 时成立。

因此方程 X=F(X)所有的解都等于 μX.F(X)，换句话说，μX•F(X)是方程唯一解。

如果我们能清楚地分辨出哪些函数是构造性的，哪些不是，则我们这个唯一解的定理的作用将大大增加。让我们定义一个非破坏性函数 G[157]，它满足的条件是

G(P)↾n = G(P↾n)↾n　　对所有 n 和 P

字母表变换就是非破坏性的，因为

f(P)↾n = *f*(P↾n)↾n

恒等函数，也是非破坏性的。任何构造性的单调函数同时也是非破坏性的。但是由于

((c → c → STOP) /< c>↾1 = c → STOP

≠ STOP

= (c → STOP) /< c>

= (((c → c → STOP↾1)/< c>)↾1

所以后继算子是破坏性的。

非破坏性函数(如 G 和 H)的任意复合仍为一非破坏性函数，因为有

G(H(P))↾n = G(H(P)↾n)↾n = G(H(P↾n)↾n = G(H(P↾n)↾n

更为重要的一点是，一个构造性函数与多个非破坏性函数的复合仍是构造性的。因此，如果函数 F,G...，H 均为非破坏性的，且它们当中有一个是构造性的，则它们复合的结果 F(G(...(H(X))...))是 X 的一个构造性函数。

以上结论可很容易地推广到含多个自变量的函数的情形。譬如说，进程的并行组合对两个自变量都是非破坏性的，因为

(P ‖ Q)↾n = ((P↾n) ‖ (Q↾n))↾n

假设 E 为含有进程变量 X 的一个表达式。如果对 X 在表达式 E 中每个出现的地方，均有一个构造性函数作用于它，而且不存在作用于它的破坏性函数，我们说 E 关于 X 是卫式(Guarded)的。例如，以下表达式关于 X 是卫式

(c → X |d→f(X ‖ P)|e → (f(X) ‖Q ‖ ((d → X) ‖ R

由此我们可以得到重要结论：函数的构造性可以仿照下面卫式表达式的语法条件定义

D1 凡仅由并发算子、符号变换算子及通用选择算子构成的表达式，是保持卫式特性的。

D2 不含 X 的表达式，是关于 X 的卫式。

---

[157] 非破坏性函数是函数语言的一个概念，指的是不改变操作对象的某种性质，例如，经过变换后，对象之前的某种属性还保持着。

D3 如果对所有 x，P(X, x)是保持卫式特性的，则下面的通用选择是关于 X 的卫式。

(x:b → P(X,x))

D4 如果 P(X)关于 X 是卫式，则符号变换 f(P(X))也是关于 X 的卫式。

D5 如果 P(X)和 Q(X)都是关于 X 的卫式，则它们构成的并发系统 P(X) ‖ Q(X)关于 X 仍是卫式。

最后，我们得到以下结论

L6 如果表达式 E 是关于 X 的卫式，则方程 X=E 有唯一解

第三章　不确定性(NONDETERMINISM)

3.1 引言(Introduction)

选择算子(x:b → P(X))用来定义一个行为可能是一个范围的进程；并发性算子‖允许某个其它进程在集合 B 提供的选择对象之中做出抉择。例如，换钱机 CH5C(1.1.3 节 X2)[158]就允许顾客选择换钱的方式，如换出三枚小硬币和一枚大硬币，或是两枚大的，一枚小的。

这类进程叫做确定性的，因为每当进程处于有多于一个的可能事件的关口，其选择总是可以通过该进程的外部环境来确定。这种确定性可以理解为是环境做出的选择，或从略微弱一点的意义上讲，环境可以在实现选择的那一时刻观察到所作的选择[159]。

有一种场景，一个进程具有一定可选范围的行为，但环境并不能影响或甚至都观察不到进程是如何在哪些可选行为之间进行选择的。例如，一台不同的换钱机可能给出上述两种换钱方法中的任意一种[160]；而到底会给出哪种，使用者[161]不能控制，甚至都无法预测。这个选择就像是在机器内部，由机器本身以任意的或非确定的方式进行的[162]。环境不能控制这个选择，甚至观察不到选择的过程；环境无法知道选择发生的精确时间，尽管环境事后可以依据进程的后续行为推断出当时的选择结果。

这种非确定性没有什么神秘的：是由于有时候我们有意要忽略影响某些选择的因素，从而这种非确定性才产生出来了。譬如说，机器换钱的各种组合方式可能依赖于机器装入大、小硬币时的状态，但我们把这些有关事件排除在进程字母表之外了。因此，在描述一些实际系统和机器的行为时，为了保持描述的高度抽象，非确定性是很有用的。

---

[158] CH5C = in5p → (outlp → outlp → outlp → out2p→ CH5C ｜ out2p → outlp → out2 → CH5C)

[159] 表达一种确定性，不会发生环境不知道(观察)一个进程下一步做什么，进程是一个黑盒子的现象。

[160] 指 CH5C 的两种找钱的方式。

[161] 使用这个换钱机的“使用者”可以理解为是这个换钱机的环境，驱动和(或)观察者这个换钱机。

[162] 或者是基于一种内部的算法，但这个算法的细节对环境不可知，不可观察。

3.2 非确定性(Nondeterministic or)

假设 P 和 Q 是进程，我们引入记号

$P \sqcap Q$　　(P 或 Q)

表示这样一个进程：它或者按 P 动作，或者按 Q 动作，而进程在两者之间的选择，是外部环境不知道或不能控制的情况下进行的。使用⊓时，我们假设各运算对象的字母表相同

$\alpha(P \sqcap Q) = \alpha P = \alpha Q$

例子

X1 一台换钱机总是按两种换钱方法中的任一种正确地换钱

$CH5D = (in5p \rightarrow ((outlp \rightarrow outlp \rightarrow outlp \rightarrow out2p \rightarrow CH5D)$

$\sqcap (out2p \rightarrow outlp \rightarrow out2p \rightarrow CH5D)))$　　□

X2 每次使用 CH5D 时，它换出的钱的组合都有可能不同。下面的一台机器，它总是按同一种组合方式换钱，但是我们不知道机器启动时会选择哪一种组合(参看 1.1.2 节 X3，X4)。令

$CH5E = CH5A \sqcap CH5B$[163]

当然了，只要这台机器送出第一枚硬币后，其后续行为就完全可以预测了，因此

$CH5D \neq CH5E$[164]　　□

通过二元算子⊓，我们以最单纯、最简单的形式引入了非确定性的概念。当然我们不打算用⊓实现一个进程。否则就好比我们建立了 P 和 Q，再把它们放到一只黑袋里，然后任意摸出一个来，把另一个扔到一边，这样做自然就愚蠢之至了！

非确定性的主要用途在于对进程描述。描述为(P⊓Q)的进程，可以通过建立 P 或 Q 来实现，两者之间的选择可由实现者根据某些与描述无关(并且是被描述有意忽略)的因素做出，如费用低廉、快速反应时间，或早出产品等因素。其实，算子⊓并不经常在描述中直接使用；非确定性通常是在使用本章后面定义的一些其它算子时自然的引出的一个性质。

3.2.1 法则(Laws)

非确定性选择要遵从的代数法则相当的简明。P 与 P 之间做选择等于不做。

---

[163] 一旦第一次的非确定性决定了是走 CH5A 或者 CH5B。之后，递归的行为就要么是 CH5A 或者 CH5B 了。是一个确定性行为了。

[164] CH5D 每次都会重复递归，持续有非确定性行为。

L1 $P \sqcap P = P$ (幂等性)

选择的先后次序无关紧要

L2 $P \sqcap Q = Q \sqcap P$ (对称性)

三个选择对象之间的选择可以拆成两个连续的二元选择。分解方式不限。

L3 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ (结合律)

做出非确定性选择的时间点不重要。一个进程先做 x 再做选择与先选择后执行 x 没有什么区别。

L4 $x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q)$ (分配律)

法则 L4 也说明前缀算子对非确定性是满足分配律的。我们把这类算子说成是可分配的。一个二元算子如果在其两个自变量位置上对⊓分别有分配律，那么我们说它是可分配的。目前为止，大多数关于进程运算的算子都是满足分配律的。它们是

L5 $(x{:}b \rightarrow (P(x) \sqcap Q(x))) = (x{:}b \rightarrow (P(x)) \sqcap (x{:}b \rightarrow Q(x))$

L6 $P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$

L7 $(P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$

L8 $f(P \sqcap Q) = f(P) \sqcap f(Q)$

但是，递归算子却不是可分配的，除非在最简单的情况下，即当⊓的两个运算对象相同时，它才满足分配律，这一点可由以下两个进程之间的差别简单加以说明。

$P = \mu X \bullet ((a \rightarrow X) \sqcap (b \rightarrow X))$[165]

$Q = (\mu X \bullet ((a \rightarrow X)) \sqcap (\mu X \bullet (b \rightarrow X))$[166]

它们的区别在于，每一次迭代时，P 都可在 a 和 b 之间独立选择，因此它的迹包括

$\langle a, b, b, a, b \rangle$

而 Q 则必须在总是执行 a 或是执行 b 之间做出选择，因此它的迹中就不可能包括上述的迹[167]。但是 P 却可能选择总是执行 a 或总是执行 b[168]，于是有

$traces(Q) \subseteq traces(P)$

在某些理论中，非确定性必须是公平的，其含义是，一个无穷次可能发生的事件最终必须发生(尽管不限它可能延迟发生的时间)[169]。但在我们的理论里没有这个公平

[165] P 进程每次完成一个非确定性选择之后，会递归到初始，下一次再次面临非确定性的选择(a 与 b 之间)，例如，CH5D。

[166] Q 进程在完成第一次非确定性选择之后，因为递归算子已经在各自的局部行为部分，所以不会再遇到开始时的非确定性算子⊓。例如，CH5E。

[167] Q 的迹要么是<a,a,a,...,a>，要么是<b,b,b...b>。不可能 a 与 b 的非确定性的交错。

[168] P 的非确定性行为存在着每次选择 a 或者 b 的可能性。环境只是不知道为什么，不可观察而已。

性的概念。由于我们只观察进程行为的有穷迹，因此如果一事件可以无限地延迟下去，我们就根本无法说清它是否要发生。如果我们想坚持使该事件最终发生，我们必须说明存在一个 n，使得每个长度超过 n 的迹都包含该事件。然后在设计进程时就要求它满足这个约束条件。例如，以下定义的进程 P_0 中，事件 a 一定会在 n 步之内发生

$$P_i = (a \rightarrow P_0) \sqcap (b \rightarrow P_{i+1})$$[170]

$$P_n = (a \rightarrow P_0)$$

以后我们会说明 Q 和 P_0 都是 P 的一个合法的实现。

如果非确定性的公平性确实需要保证，就应对非确定性部分单独地进行描述和实现，例如规定非确定性选择的每种选择的概率不为零，看来将复杂的概率推理和讨论进程行为的逻辑正确性分别处理，应该是很合理的。

根据 L1 至 L3，可引入一个多重选择算子[171]，它会是很有用的。设 S 为一有限的非空集合

$$S = \{i, j, ...k\}$$

那么我们定义

$$\sqcap_{x:S} P(x) = P(i) \sqcap P(j) \sqcap ... \sqcap P(k)$$

当 S 为空集或无穷集时，⊓无意义。

3.2.2 实现(Implementations)

如上所述，引入非确定性的一个主要原因是为了摆脱实现细节，进行抽象描述。这意味着一个非确定性进程 P 可能有很多种不同的实现方式，每一种实现反映进程的一种可观察到的行为特征。这些相异性是由于对进程内在的非确定性允许有不同的处理方法而引起的。有关的选择可以由实现者在进程启动前进行，也可以等到进程实际运行时再进行。例如 (P ⊓ Q)的一种实现方法是选择第一运算对象

$$or\ 1(P, Q) = P$$

另一种方法是选择第二个运算对象，例如，这样做在某个具体机器上可能会更有效

$$or\ 2(P, Q) = Q$$

---

[169] 这种可能发生的一定会发生的逻辑通常叫做 Liveness。可参阅时序逻辑(Temporal Logic)。

[170] a 事件可能在任意步发生，但无法保证。非确定性的“算法”不可知。

[171] 可以理解为多重非确定性选择符号。

而第三种方法是把选择推迟到进程运行时再进行，由环境选中一个事件。如果这个事件只对两个进程中的一个是可能的，那么就选那个可能的进程。如果该事件对两个进程都是可能的，则再推迟选择

```
or3(P, Q) = λx • if P(x) = "BLEEP then
        Q(x)
        else if Q(x) = "BLEEP then
                (P(x)
                else
                        or3(P(x), Q(x))
```

这里对这个算子我们给出了三种不同的可能实现的方法。其实远不止这三种；譬如，有一种实现的前五步可以似 or3 动作；如果经过五步仍无法选定 P 或 Q，那么它就任意地选择 P。

由于(P ⊓ Q)的设计无法控制对 P 或对 Q 的选择，因此他就必须保证他设计的系统对这两种选择都能正常工作。如果 P 或 Q 中有一个与其环境有发生死锁的危险，则(P ⊓ Q)也一定有此危险。实现方法 or3 只是推迟选择，最终推给环境去做，让它在 P 和 Q 中选择不死锁的进程，从而减小死锁的危险，由于这个原因，or3 的定义有时被称做安琪儿非确定性。但是从效率角度看，这种做法的代价太大了：如果第一步时，没能在 P 和 Q 之间做出选择的话，P 和 Q 就必须并发执行，这种并发现象要一直继续到环境选择了一个只对它们中的一个有可能的事件。在非常简单但又极端的情形 or3(P, P)中，不可能选到只对其中一个进程有可能的事件，这时这种实现是极端的低效的。

与 or3 相比，or1 和 or2 是非对称的，因为

```
or 1(P, Q) ≠ or1(Q, P)
```

这看来好象违反了 3.2.1 节的 L2；其实不然。因为法则是应用于进程的，而不是应用于进程的任意一种具体实现的。事实上每个法则断言等式左右两边进程的所有实现集合恒等。例如，由于 or3 是对称的，所以

```
{orl(P, Q), or2(P, Q), or3(P, Q)}
= {P, Q, or3(P, Q)}
= {or2(Q, P), orl(Q, P), or3(Q, P)}
```

引入非确定性的一大好处是，能避免在选用上述两种实现时，所引起的对称性的丧失，还能避免选用对称实现 or3 的低效率问题。

3.2.3 迹(Traces)

设 s 为 P 的一个迹，则 s 也可能是(P ⊓ Q)的一个迹，因为 P 可能被选中。同理如果 s 为 Q 的一个迹，那它也可能是(P ⊓ Q)的一个迹。反过来说，(P ⊓ Q)的每个迹肯定是两个选择对象中某一个的迹，或是它们的公共迹。因此，在执行完 s 之后，(P ⊓ Q)的行为就要由执行 s 的那一个 P 或 Q 所定义；如果两者都能执行 s，那么该选择仍是非确定的。

Ll traces(P ⊓ Q) = traces(P) ∪ traces(Q)

L2 (P ⊓ Q)/s = Q/s

= P/s

= (P/s) ⊓ (Q/s)

3.3 一般选择(General choice)

由于(P ⊓ Q)的环境不能对在 P 与 Q 之间的选择实行控制，也不知道选择是怎样进行的，甚至不知道选择是在何时发生的。因此由(P ⊓ Q)生成的进程的组合就不是很有用，因为环境必须随时准备好应付 P，或者应付 Q；而只需要分别应付它们中的一个就简单很多。为此，我们引入另一个操作(P □ Q)，如果环境从它执行第一个动作开始就可以通过这个□操作实行对选择的控制，那么环境就能控制(P □ Q)，选出和感知 P 或者 Q。如果第一个动作不是 P 的可能动作，就选择 Q；反之，就选择 P。但是，如果该动作对 P 和 Q 都能使可能的，那么它们之间的选择为非确定的。(当然，对 P 和 Q 都不可能的事件是不可能发生的。) 如同往常一样，进程的字母表的关系如下：

α(P□Q) = αp = αQ

如果 P 的所有初始事件对 Q 均是不可能的，在这种情况下，一般选择算子就和选择算子|完全一样了，后者是一直被用来表示两个不同事件之间的选择的。故

(c → P □ d→ D) = (c → P | d→ Q)　　若 c≠d

然而如果 P 和 Q 的初始事件都一样的话，(P□Q)就退化为非确定选择，即

(c → P□c → Q) = (c → P ⊓ c → Q)

这里我们采用了这样一个约定：→的结合度高于□。

3.3.1 法则(Laws)

有关□的代数法则与⊓的类似，因此

L1-L3 是幂等的、对称的、结合的。

L4 P ⊓ STOP = P

以下法则将该操作的非形式定义形式化表述

L5 (x:a → P(x)) □ (y:b → Q(y)) =

(z:(A ∪ B) →

(if z ∈ (A-B) then P(z)

else if z ∈ (B-A) then Q(z)

else if z ∈ (A∩B) then (P(z) ⊓ Q(z)))

象所有到目前为止我们引入的其它算子(递归式除外)一样， □对⊓也有分配律，即

L6 P □ (Q ⊓ R) = (P □ Q) ⊓ (P □ R)

但是更令人吃惊的是，⊓对□也有分配律

L7 P ⊓ (Q □ R) = (P ⊓ Q) □ (P ⊓ R)

这条法则是说，进程非确定地做出的选择与环境做出的选择互相独立，也就是进程和环境中不论哪个所做的选择都不影响另一方的选择。假设约翰是做非确定选择的代理人，玛丽是外部环境。在法则中等式左边，玛丽在 Q 和 R 之间做选择即□，而约翰在 P 和玛丽做选择的结果之间做选择即⊓。等式右边，玛丽可做的选择如下

(1) 让约翰在 P 和 Q 之间选择

(2) 让约翰在 P 和 R 之间选择

如果约翰选择 P，那么 P 就将是方程两边的结果。但如果约翰没选择 P，那么 Q 和 R 之间的选择就要由玛丽来做。这样，法则中等式两边所描述的两种选择对策的结果都是一样的。当然，这一推理同样适用于 L6。

以上给出的解释多少有一点难以捉摸；如果把这条法则解释为本章以后要给出的那些更为明了的定义及法则的一个意想不到的而又不可避免的推论，也许会令人更好理解些。

3.3.2 实现(Implementation)

这种选择算子的实现有 L5 即可得到。假设下述定义中的 or 是对称的，则算子 choice 也是对称的

choice(P, Q) =λx.if P(x) = "BLEEP then Q(x)

else if Q(x) = "BLEEP then P(x)else or(P(x), Q(x))

3.3.3 迹(Traces)

(P □ Q)的每个迹一定是 P 或 Q 的一个迹，反之亦成立

Ll traces(P □ Q) = traces(P) ∪ traces(Q)

下一条法则与有关⊓的对应法则稍有差别

L2 (P □ Q)/s = P/S　　如果 s∈ traces(P)-traces(Q)

= Q/s　　如果 s∈ traces(Q)-traces(P)

=(P/s) ⊓ (Q/s)　　如果 s≠<>且 s∈ traces(P) ∩ traces(Q)

3.4 拒绝集(Refusals)

(P ⊓ Q)与(P □ Q)之间的区别很微妙。不能根据它们的迹把它们区分开，因为它们中任一个的迹同时也可能是另一个进程的迹。但是如果把它们放到某个特定的环境中，在这个环境中，(P ⊓ Q)一开始就可能死锁，而(P □ Q)没有死锁，例如设 x≠y，并且令

P = (x → P), Q=(y → Q), αp = αQ = {x,y}

则(P □ Q) ‖ P = (x → P) = P[172]

而(P ⊓ Q) ‖ P = (P ‖P) ⊓ (Q‖ P)[173]

= P ⊓ STOP

这说明在环境 P 中，(P ⊓ Q)可能会达到死锁，但(P □ Q)不可能。当然，即使对(P ⊓ Q)而言，也不能说死锁就一定会发生；而且如果一开始没发生死锁的话，我们以后都无法知道其实它本来是可能发生的。不过仅就死锁发生的可能性这一点，就足以把(P ⊓ Q)和(P □ Q)区分开了。

一般说来，假设 X 是一进程 P 的环境最初提供的事件的一个集合，我们假设环境与 P 具有相同的字母表。如果把 P 放到这个环境里，在开始执行时，就有可能发生死锁的话，我们就说 X 是 P 的一个拒绝集。P 的这类拒绝集的集合记为

refusals(P)

---

[172] (P □ Q) ‖ P = (x → P | y → Q) ‖ P = (x → P | y → Q) ‖ (x → P) = x → P = P。可参阅 2.2.1 L4 的证明。

[173] 通过并发算子‖对非确定性 ⊓满足分配律的性质。

值得注意的是一个进程的拒绝集的集合构成了一个符号事件的集合。这确实是有点太复杂了，但为了适当处理非确定性问题，好象也确实是不可避免的。相比较而言，使用表示进程可以接受的事件的符号集合，称作就绪集，好象要比用拒绝集更自然些；然而用拒绝集要更简单些，因为拒绝集服从法则 L9 及 L10(这两条法则将在 3.4.1 节给出)，而就绪集合的法则要更复杂些。

引入了拒绝集的概念，我们就可以清晰地、形式化地区分确定性进程与非确定性进程了。一个进程，如果它不能拒绝执行它能执行的任何事件，我们就说它是确定性的。换句话说，一个集合为某个确定性进程的拒绝集，当且仅当该集合不包含进程最初能执行的事件；或更形式地记为

(P 是确定性进程) ⇒ (X ∈ refusals(P) = (X ∩ P^0 = {}))

其中 P^0 = {x| <x>∈ traces(P)}

这个条件不仅对进程 P 的第一步执行起作用，而且也适用于 P 执行了任何可能的动作序列后的行为。于是我们可以定义

(P 是确定性进程) ≡ ∀s：traces(P) • (X ∈ refusals(P/s) ≡ (X ∩ $(P/s)^0$={}[174]))

不具备这个性质的进程就是非确定性进程，也就是说，在某个时刻，进程可以执行某个事件，但它也可能(由于某个内部非确定性选择)拒绝执行这一事件，即使其环境已做好了准备要执行这一事件。

3.4.1 法则(Laws)

以下法则定义了各种简单进程的拒绝集。进程 STOP 什么都不做，即拒绝一切事件

L1 refusals($STOP_A$) = A 的所有子集 (包括 A 本身)

进程(c → P)拒绝不包含事件 c 的任何事件集合，即

L2 refusals(c → P) = {X| X ⊆ (αP - {c})}

这两条法则有一个共同的通用形式

L3 refusals(x:b → P(x)) = {X | X ⊆ (αP-B)}

如果 P 能拒绝 X,则(P ⊓ Q)也能拒绝 X,因为(P ⊓ Q)可非确定性的选择 P。同理 Q 的每个

[174] X ∩ $(P/s)^0$={}的含义是，如果 X 是一个事件的集合，X 属于 refusals(P/s)，其中包含了至少一个会使得(P/s)进入死锁的事件。但令人高兴的是，X 中的任何一个事件都不属于 P/s 进程可以响应的初始事件。换言之，这个 X 不会影响 P/s 进程的走向。

拒绝集也是(P ⊓ Q)的可能的拒绝集。而这拒绝集的合集就是(P ⊓ Q)所有的拒绝集，所以

L4 refusals(P ⊓ Q) = refusals(P) ∪ refusals(Q)

(P □ Q)的拒绝集与上面的正好相反。如果 X 不是 P 的一个拒绝集，则 P 就不能拒绝 X，因此(P □ Q)也就不能拒绝 X。同理如果 X 不是 Q 的一个拒绝集，那么它也就不是(P □ Q)的一个拒绝集。然而如果 P 和 Q 都能拒绝 X 的话，(P □ Q)才可以拒绝 X。即

L5 refusals(P □ Q) = refusals(P) ∩ refusals(Q)

比较 L5 和 L4，就能看出 □ 与⊓之间的区别了。

如果 P 能拒绝 X，Q 能拒绝 Y，那么它们的组合(P ‖ Q)不仅能拒绝被 P 拒绝的所有事件，而且能拒绝被 Q 拒绝的所有事件，也就是说，它能拒绝这两个集合 X 和 Y 的和集，即

L6 refusals(P ‖ Q) = {X ∪ Y | X ∈ refusals(P) ∧ Y ∈ refusals(Q) }[175]

对于符号变换，有关的法则是很清楚的

L7 refusals(*f*(P)) = { f(X) | X ∈ refusals(P) }

还有几个关于拒绝集的一般法则。进程只能拒绝在自己的字母表内的事件。当进程的环境不提供任何事件时，进程死锁；如果一个进程拒绝一个非空集合，也拒绝该集合的任何子集[176]。最后，一开始不能发生的事件 x 可以被加到已被拒绝的集合 X 中去。

L8 X ∈ refusals(P) ⇒ X ⊆ αP

L9 { } ∈ refusals(P)

L10 (X∪Y) ∈ refusals(P) ⇒ X ∈ refusals(P)

L11 X∈refusals(P) ⇒ (X∪{x}) ∈ refusals(P) ∨ <x> ∈ traces(P)　　x∈αP

---

[175] (P ‖ Q)与(P ⊓ Q)的拒绝集貌似类似，但其实是不一样的。例如，假设，符号表是{a,b,c,d}，P = (a → P | b → P)，Q = (b → Q | c → Q)。因此 P 的拒绝集合 refusals(P)={{c},{d},{c,d}}；Q 的拒绝集合 refusals(Q)={{a},{d},{a,d}}。refusals(P ⊓ Q) = refusals(P) ∪ refusals(Q) = {{a},{c},{d},{a,d},{c,d}}。refusals(P ‖ Q) = refusals(b → (P ‖ Q))={X ∪ Y | X∈ refusals(P) ∧ Y∈ refusals(Q) } = {{a},{c},{d},{a,c},{a,d},{c,d},{a,c,d}}。显然，(P ‖ Q)拒绝一个提供{a,c,d}事件的环境，但(P ⊓ Q)可以很轻松的接受提供{a,c,d}事件的环境。{a,c,d}和子集是(P ‖ Q)的一个 refusal，但不是(P ⊓ Q)的 refusal。

[176] 这个结论很明显和重要，例如，假设符号表是{a,b,c,d,e}，对于一个进程 P = e→ P(首发事件必须是 e)。如果无法接受(拒绝)事件集合{a,b,c,d}在最初始，显然也无法接受{a,b,c,d}的任何子集。

3.5 屏蔽(Concealment)

一般说来，一个进程的字母表仅包含那些我们认为是与进程有关的事件，而且它们的发生还需环境的同时参予。在描述一个机械装置的内部行为时，我们经常需要考虑那些代表该装置内部转换的事件。这类事件可能代表着构成这个机械装置的各个并发组成部分之间的相互作用和通信联系，例如 CHAIN2(2.6 节 X4)和 2.6.2 节中 X3。这类装置构成之后，我们屏蔽掉它的组成部分的结构细节；并且还希望屏蔽掉装置内部发生的所有动作。实际上，我们是要这些动作自动发生，而且以最快的速度进行，而不被进程的环境观察到或是受其控制。设 C 是用这个办法屏蔽掉的事件的一个有限集合，那么

$P \setminus C$

即为一个似 P 动作的进程，只是在 C 中的任何事件的发生都被屏蔽掉了。显然我们希望有 $\alpha(P \setminus C) = (\alpha P)-C$

例子

X1 我们可以把一台很吵闹的自动售货机(2.3 节 X1)放到一个隔音箱里，就能消除噪声

$NOISYVM \setminus \{clink, clunk\}$

该售货机出售太妃糖的本领从没用上，也可以把这一功能从它的字母表中抹去，而不影响它的实际使用。最后得到的进程就等于简单自动售货机了。

$VMS= NOISYVM \setminus \{clink, clunk, toffee\}$

当两个进程被组合到一起并发运行时，我们通常把它们之间的相互作用看作是它们所组成的系统的内部动作；我们希望这种相互作用自主发生，而且速度越快越好，而不让系统的外部环境感知，也不让环境对它们进行干预。因此需要屏蔽的正是属于这两个进程字母表的交集中的符号。

X2 设

$\alpha P=\{a,c\} \quad P=(a \rightarrow c \rightarrow P)$

$\alpha Q=\{b,c\} \quad Q=(c \rightarrow b \rightarrow Q)$ (参见 2.3.1 节 X1)

动作 c 同在 P 和 Q 的字母表中，因此它被看作是一个内部动作而被屏蔽，有

$$(P \parallel Q) \setminus \{c\} = (a \rightarrow c \rightarrow \mu X \bullet (a \rightarrow b \rightarrow c \rightarrow X \mid b \rightarrow a \rightarrow c \rightarrow X)) \setminus \{c\}$$
$$= a \rightarrow \mu X \bullet (a \rightarrow b \rightarrow X \mid b \rightarrow a \rightarrow X)$$

3.5.1 法则(Laws)

前几条法则将陈述屏蔽一个空集毫无效用，而且集合中符号被屏蔽的先后次序也无影响。这组法则中其余的说明屏蔽对其它算子有分配律。

什么都不屏蔽的话，就等于将一切暴露给环境，即

L1 $P \setminus \{\} = P$

一个个地屏蔽和将它们同时屏蔽是一回事

L2 $(P \setminus B) \setminus C = P \setminus (B \cup C)$

屏蔽对非确定性选择有分配律

L3 $(P \sqcap Q) \setminus C = (P \setminus C) \sqcap (Q \setminus C)$

屏蔽不影响已停止的进程的行为，只影响它的字母表

L4 $STOP_A \setminus C = STOP_{A-C}$

屏蔽的目的在于让任何被屏蔽的事件自动地在一瞬间发生，而且要使外界完全看不到它们的发生。没被屏蔽的事件仍然不变。故有

L5 $(x \rightarrow P)\setminus C$

$= x \rightarrow (P\setminus C)$ 若 $x \in C$

$= (P\setminus C)$ 若 $x \in C$

如果 C 只包含 P 和 Q 各自独立参予的事件，则 C 的屏蔽对 P 和 Q 的并发组合有分配律

L6 设 $\alpha P \cap \alpha Q \cap C = \{\}$

那么 $(P \parallel Q) \setminus C = (P \setminus C) \parallel (Q \setminus C)$[177]

这条法则不是很有用，因为我们通常是希望把并发进程间的相互作用屏蔽掉，也就是说屏蔽掉属于 $\alpha P \cap \alpha Q$ 的事件，这类事件的发生是需要这两个并发进程共同参加的。

屏蔽显然对 1-1 符号变换函数有分配律

L7 $f(P \setminus C) = f(P) \setminus f(C)$

如果供选择的可能初始事件都没被屏蔽，那么屏蔽后的初始选择扔与屏蔽前的一样

L8 若 $B \cap C = \{\}$

那么 $(x{:}b \rightarrow P(x)) \setminus C = (x{:}b \rightarrow P(x) \setminus C))$

---

[177] 如果要屏蔽的事件不属于两个进程都同时感知的，可以“单独”在各个进程的行为体里忽略。否则，会改变这个并发进程组合进程的行为。例如，$(x \rightarrow y \rightarrow P) \parallel (x \rightarrow z \rightarrow Q) \setminus \{x\}$，如果先应用{x}的过滤，并发组合进程变成 $(y \rightarrow P) \parallel (z \rightarrow Q)$，并为一个 STOP 进程了。正确的屏蔽行为应该是$(x \rightarrow y \rightarrow P) \parallel (x \rightarrow z \rightarrow Q) \setminus \{x\} = (x \rightarrow (y \rightarrow P) \parallel (z \rightarrow Q)) \setminus \{x\} = (y \rightarrow P) \parallel (z \rightarrow Q)$。

象选择算子□一样，屏蔽会引入非确定性。当几个不同的被屏蔽的事件都能够发生时，我们不能确定它们中的哪个将要发生；但不论哪个真发生了，都被屏蔽掉。即

L9 如果 B ⊆ C，且 B 是非空的有限集合

那么$(x : b \to P(x)) \setminus C = \sqcap_{x \in B} (P(x) \setminus C)$[178]

在一种折中的情形中，即当初始事件中有一部分被屏蔽，有一部分没被屏蔽时，情况就要更复杂些 。考虑如下进程

$$(c \to P \mid d \to Q) \setminus C \quad c \in C, d \in C$$

屏蔽的事件 c 成为进程的内部动作，可以自动地而且立刻的发生，这导致进程的整体行为是$(P \setminus C)$，不存在事件 d 发生的概率。另外，我们不能保证 d 不会发生。如果环境就绪，事件 d 可能可以在隐含事件(c)发生之前发生；如果这种情况，c 就不会再发生了。但另外一种关于 d 事件的情况是，d 事件触发或者发生了，但是是在隐含事件发生后的(P\C)阶段触发的[179]。因此，关于 d 事件的两部分行为组成的整体行为是：

$$(P \setminus C) \Box (d \to (Q \setminus C))$$

上面关于 d 事件的行为与关于 c 事件被屏蔽后的(P\C)行为是非确定性的关系。这部分是一个比较绕人的复杂定理，可以总结如下：

$$(c \to P \mid d \to Q) \setminus C = (P \setminus C) \sqcap ((P \setminus C) \Box (d \to (Q \setminus C)))$$[180]

类似的推理可适用在一个更普遍的法则

L10 如果 C ∩ B 是非空有限集合，那么我们有：

$$(x:b \to P(x)) \setminus C = Q \sqcap (Q \Box (x:(B-C) \to P(x)))$$

其中

$$Q = \sqcap_{x \in B \cap C} P(x) \setminus C$$

---

[178] 假设进程(x:b → P(x))叫做 Q。对其 init 事件的过滤，导致其行为树出现众多的非确定性选择，在外界不可观察的情况下，进程行为通过非确定性的方式进入到 P(x)\{C}部分，并形式化描述为$\sqcap_{x \in B}$ (P(x)\C)。可参阅图 3.3。

[179] 这个定理比较难理解。英文原著是“The concealed event *c* may happen immediately. In this case the total behavior will be defined by *(P \ C)*, and the possibility of occurrence of the event *d* will be withdrawn. But we cannot reliably assume that *d* will not happen. If the environment is ready for it, *d* may very well happen before the hidden event, after which the hidden event *c* can no longer occur. But even if *d* occurs, it might have been performed by *(P \ C)* after the hidden occurrence of *c.*”。这里面最关键的是要注意对 d 事件的观察或者触发，是可能发生在(P\C)阶段，因此要通过 General Choice(□)来覆盖((P\C) □ (d → (Q \ C)))。

[180] 这个定理乍看之下应该是等于(P \ C) ⊓ (d → (Q \ C))。这是不正确的。例如，假设这个复合进程非确定性的选择了(P\C)的路径，假设(P\C) = (d → Z)。如果这个时候发生事件 d， 环境还是可能选择非确定性的触发 d → (Q\C)的。

以上相关法则的图示说明将在 3.5.4 节中给出。

要注意的是，\C 对□并不能后溯分配。我们举个反例即可说明

$(c \rightarrow STOP \square d \rightarrow STOP) \setminus \{c\}$

$= STOP \sqcap (STOP \square (d \rightarrow STOP))$ L10

$= STOP \sqcap (d \rightarrow STOP)$ 3.3.1 节 L4

$\neq d \rightarrow STOP$

$= STOP(d \rightarrow STOP)$

$= ((c \rightarrow STOP) \setminus \{c\} \square ((d \rightarrow STOP) \setminus \{c\})$

屏蔽会使进程的字母表缩小，与之相对应我们还可以再定义一个使进程 P 字母表扩大的操作，扩大的方法是把某个集合 B 中的符号并到 P 的字母表中来。令

$\alpha(P_{+B}) = \alpha P \cup B$

$P_{+B} = (P \parallel STOP_B)$ $B \cap \alpha P = \{\}$

集合 B 中的事件是不会实际发生的，所以 P_{+B}的行为实质上与 P 的行为是一样的，即有

L11 $traces(P_{+B}) = traces(P)$

由此推得，屏蔽\B 与用 B 扩大字母表互为逆运算

L12 $(P_{+B}) \setminus B = P$

在这里我们要适时地提出一个问题，这个问题会在 3.8 节解决。一些简单情况下，屏蔽对递归式是有分配律的。如

$(\mu X: A \bullet (c \rightarrow X)) \setminus \{c\}$

$= \mu X:(A-\{c\} \bullet ((c \rightarrow X_{+\{c\}}) \setminus \{c\})$

$= \mu X:(A-\{c\} \bullet X$ L12, L5

因此，屏蔽连续发生的无穷多个事件的序列，会导致无穷循环或是非卫式的递归式这种不好的结果。这个现象统称为进程的发散(Divergence)[181]。

即使一个发散进程可以无限多次(Infinite Often)的机会执行没有被屏蔽的事件，以上进程行为发散的问题还是会出现的。例如

$(\mu X \bullet (c \rightarrow X \square d \rightarrow P)) \setminus \{c\}$

$= \mu X \bullet ((c \rightarrow X \square d \rightarrow P) \setminus \{c\}$

$= \mu X \bullet (X\setminus\{c\}) \sqcap ((X\setminus\{c\}) \square d \rightarrow (P\setminus\{c\}))$ L10

这里的递归式是非卫式的，因而趋于发散。尽管环境似乎可以无止境地提供选中事件 d

[181] 对递归的进程，如果屏蔽了卫式(Guarded)的符合(事件)，例如，(tick → CLOCK) \{tick}，我们得到一个发散的 CLOCK = CLOCK。这个递归进程没有解，是发散的序列。丢失了 Guarded 的特性。

的这种可能机会，却无法阻止进程去执行被屏蔽的事件，而不执行 d。但是这一可能性似乎有助于得到最高效的实现方法。在 3.2.1 节中我们不坚持要求非确定性进程中各种选择的公平性的决定与之有一定关系。在 3.8 节中，我们还要更严格地讨论发散的问题。

从某种意义上而言，有一种感觉，屏蔽实际上是公平的。例如，假设 $d \in \alpha R$，考虑进程

$$((c \to a \to P \mid d \to STOP) \setminus \{c\} \parallel (a \to R)$$

$$= ((a \to P \setminus \{c\} \sqcap (a \to P \setminus \{c\} \Box d \to STOP)) \parallel (a \to R) \quad L10$$

$$= (a \to P \setminus \{c\}) \parallel (a \to R) \sqcap (a \to P \{c\} \Box d \to STOP) \parallel (a \to R)$$

$$= a \to ((P \setminus \{c\}) \parallel R)$$

这说明一个进程，尽管它可以提供环境[182]选择被屏蔽动作 c 或非屏蔽动作 d，但它却不能坚持非要非屏蔽事件发生不可。如果环境(在本例中是 $a \to R$)没有为 d 的发生做好准备，那么被屏蔽的事件 c 就必须发生，从而环境有机会与屏蔽后的进程结果行为(即$(a \to P \setminus \{c\})$)发生相互作用。

3.5.2 实现(Implementation)

为简便起见，我们实现一次只屏蔽一个符号的进程，令

$hide(P, c) = P \setminus \{c\}$

对含两个或两个以上符号的集合，进程可以将其符号一个接一个地屏蔽起来，原因很简单，因为有

$$P \setminus \{c1, c2... , cn\} = \{...(P \setminus \{c1\} \setminus \{c2\}) \setminus ... \setminus \{cn\}$$

最简单的实现方法是让被屏蔽的事件在可能发生时，以最快的速度悄悄地发生，即

```
hide(P, c) = if P(c) = "BLEEP then[183]
        (λx • if P(x) = "BLEEP then "BLEEP [184]
                                else hide(P(x), c))
    else hide(P(c), c)[185]
```

[182] 任何一个并发的进程组合，彼此是对方的环境。互相牵制，互相影响。整体的行为是互相协调才能决定的。例如，(P ‖ Q)。Q 是 P 的环境；P 是 Q 的环境。

[183] P(c) = "BLEEP 是判断 c 是否是 P 的 init event 之一。如果等于"BLEEP，说明不接受 c，那么通过下面的函数处理，例如，接着看后续的 event(hide(P(x), c))是否能够屏蔽 c，对于已经是叶子节点的 event，简单的处理为：if P(x) = "BLEEP then "BLEEP。

[184] 这个点是屏蔽算法作用在一个进程上的停止点，意味着已经没有事件可以处理了。

我们来讨论当这个屏蔽函数作用于某一个能够连续执行无穷多个被屏蔽事件的进程时，将会发生什么样的结果，例如

hide(μX • (c → X □ d → P), c)

在这种情况下，检验(P(c)= "BLEEP)时，总是产生 FALSE，所以函数 hide 就总是选择 else 子句，随后立即执行递归调用。这个递归循环永远无法退出，因而也就不可能与外界发生进一步的通信。这是如果实现了一个发散进程后带来的行为的副作用。

另外，因为屏蔽符号的顺序很重要，屏蔽运算的实现不严格遵从 L2[186]；例如，

P = (c → STOP | d → a → STOP)

那么

hide(hide(P, c), d)
= hide(hide(STOP, c), d)
= STOP

但是

hide(hide(P, d), c)
= hide(hide((a → STOP), d), c)
= a → STOP

但是正如我们在 3.2.2 节中解释过的那样，一个非确定性算子的具体实现不一定非要遵守非确定性的一般法则。上述的两个结果都可以看作是作为同一进程的两个可以接受的实现。因此，

P \ {c, d} = (STOP ⊓ (a → STOP))

3.5.3 迹(Traces)

如果 t 是 P 的一个迹，我们只要将 t 中出现的 C 的符号统统去掉，就能得到相应的 P \ C 的迹。同理，P \ C 的每个迹一定是由 P 的某个迹经过屏蔽运算得到的。相关规则定义如下

L1 如果∀s: traces(P) • ¬diverges(P/s, C)

则 traces(P \ C) = {t ↾(αP-C) | t ∈ traces(P)}

条件 diverges(P, C)的含义是：只要 C 的事件被屏蔽，进程 P 就会立刻进入发散状态，例如，执行一个由被屏蔽事件组成的，无休止的事件序列。因此我们定义发散进程如下:

---

[185] 当前这一层的 event 有 c，屏蔽 c，并接着处理后续的行为体：hide(P(c), c)。

[186] L2：(P \ B) \ C = P \ (B ∪ C)的意思是进程屏蔽 B 然后屏蔽 C，等价于屏蔽{B,C}。但没有强调屏蔽的顺序是否是 B 然后 C，还是先 C 在 B。

diverges(P, C) = ∀ n • ∃ s : traces(P) ∩ C* • #s＞n[187]

与 P \ C 的一个迹 s 相应的，可能有多个 P 的可能行为的迹 t，这些迹经过屏蔽运算后结果都一样，即 t ⍀(αP-C) = s。故继 s 之后, (P \ C)的后续行为也不一定能由 P 的某一个可能的后续行为来完全确定，法则定义为

L2 T 为有穷集合，且 s ∈ traces(P \ C)则

(P\C)/s = ($\sqcap_{t\in T}$ P/t) \C

其中 T= traces(P) ∩ { t | t ⍀ (aP - C) = s }

这些法则仅适用于不发散的进程。这种限制无伤大雅，因为我们在定义进程过程中，总是设法避免发散现象的。在 3.8 节中我们将更全面的来考虑进程的发散性。

3.5.4 示意图(Pictures)

非确定性选择可以通过在示意图中用这样一种节点来表示，由这种节点出发引出了两个或两个以上不带标记的箭头；当执行到该节点时，进程路径的选择是非确定的，外界不曾觉察或者控制的，进程就沿着某个引出的箭头向下通行。

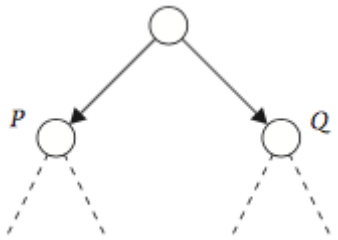


图 3.1

因此 P ⊓ Q 表示为图 3.1[188]。关于如何推断不同示意图之间的等效性的非确定性的代数法则，例如，⊓的结合律可以表示为图 3.2[189]。

[187] 定义为：任取一个整数 n，都可以找到一个 s，s 为一个被屏蔽事件组成的迹，而且 s 的长度大于 n。因此，从代数的角度，我们形式的定义出一个屏蔽一些事件后的进程进入发散状态。

[188] 图 3.1 是 P ⊓ Q 的基本模型

[189] 图 3.2 为 P ⊓ (Q ⊓ R) = P ⊓ Q ⊓ R

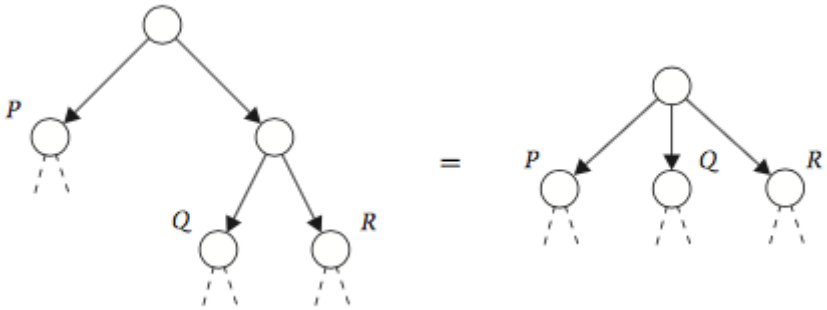


图 3.2

符号屏蔽的运算可以被看作是这样一种操作，它把要屏蔽的符号从标记中抹去，因此所有用这些符号作为标记的箭头都变成无标记的箭头。如图 3.3 所示，这样做的结果自然就会产生相应的非确定性的边。

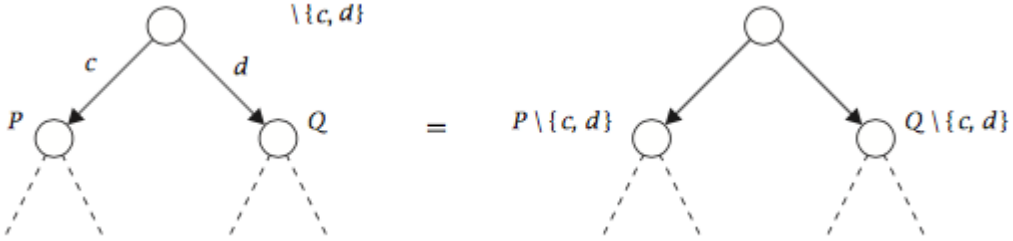


图 3.3

但是如果一个节点的有些弧有标记，而另一些没有，那么这种节点的含义是什么呢？这个问题的答案由 3.5.1 节中 L10 给出。如果我们把示意图重画成如图 3.4 的形式，就可以将这样的不对称的节点消去。

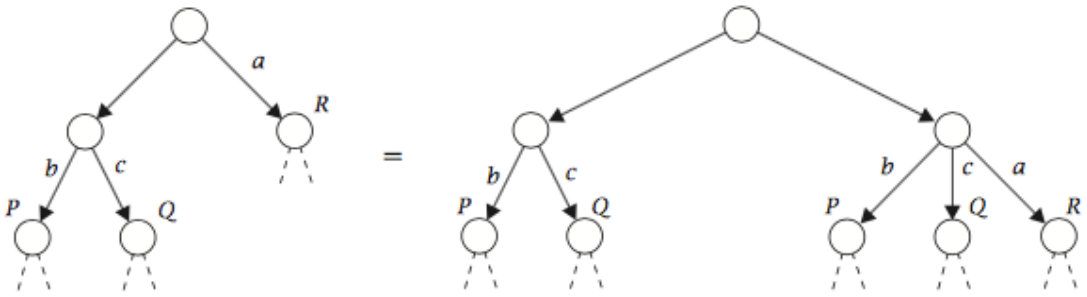


图 3.4

这种消去法对有穷树形图显然总是可行的。而对无穷示意图，只要它不包含无标记箭头构成的无穷路径，该消去法也是可行的。对含有无标记的无穷路径，例如图 3.5 所示，上述消去法则是行不通的，但这种图形只在进程发散的情况下才出现，而我

们已决定把发散看做是一种错误的进程定义。

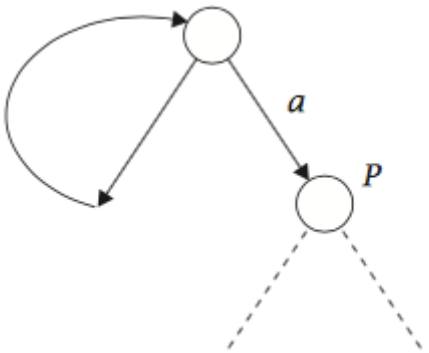


图 3.5

使用变形法则 L10，可能导致一个节点具有两个标记相同的引出线。消除这类节点可用 3.3 节结束部分给出的法则(见图 3.6)。

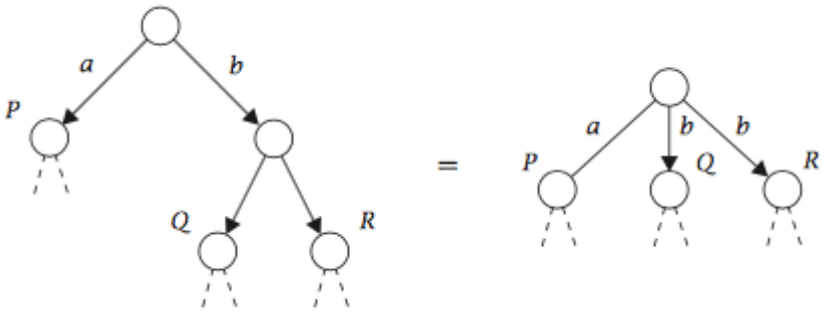


图 3.6

用图示法表示进程及它们所遵从的法则，是为了帮助记忆和理解；而不是用于大规模进程的实际变形处理。

3.6 交叉(Interleaving)

在第二章中给出的并发算子‖的定义里，对于同属于两个运算对象(进程)的字母表中的事件，这些动作(事件)的发生需要两个运算对象同时参加[190]，而系统的其余属于各自进程独有的事件则是可以任意交叉顺序发生的。通过这个算子，就能把互相作用的，但字母表不同的进程合并成并发动作的系统，而且，不会引人非确定性。

然而，有时把字母表相同的进程并在一起，使它们不直接相互作用或不需要强

[190] 这是并发算子的最本质的要求。如果一个事件属于并发的两个进程。这个事件的发生必须并发的两个进程都一致性的处理，否则环境观察不到一个正确的迹，无法记录下来。

同步的并发操作[191]，这样做有时也是很有用的。在这种情况下，系统的每个动作是系统中某个进程的单独动作。如果有一个进程不能执行某个系统动作，那么该动作一定是另一进程执行的；但是如果两个进程都有可能执行这同一个动作，则两个进程之间的选择就是非确定性的。我们把这种组合运算记为

P ||| Q　(读作 P 与 Q 交叉)

且其字母表一般约定为

$\alpha(P ||| Q) = \alpha P = \alpha Q$

例子

X1 一台自动售货机至多能连续接受两枚硬币，也至多能连续送出两块巧克力(参看 1.1.3 节 X6)

(VMS ||| VMS) = VMS2　□

X2 把四个男仆当作一个仆人使用，每次一个男仆只为一位哲学家服务(参看 2.5.4 节)　□

3.6.1 法则(Laws)

L1-3 |||满足结合律、对称律，对⊓还有分配律

L4 P ||| STOP = P

L5 当 P 不发散时，有 P ||| RUN = RUN

L6 $(x \to P) ||| (y \to Q) = (x \to (P ||| (y \to Q)) \Box y \to ((x \to P) ||| Q))$[192]

L7 如果 $P = (x{:}\ a \to P(x))$，$Q = (y{:}\ b \to Q(y))$

那么 $P ||| Q = (x{:}\ a \to P(x) ||| Q) \Box y{:}\ b \to (P ||| Q(y)))$

注意|||对□是不满足分配律的。我们举一个反例即可说明(设 b≠c)

$((a \to STOP) || (b \to Q \Box c \to R)) / <a>$

$= (b \to Q \Box c \to R)$

$\neq ((b \to Q) \sqcap (c \to R))$

$= ((a \to STOP \Box b \to Q) ||| (a \to STOP \Box c \to R)) / <a>$

公式串的最左端的公式中，事件 a 的发生只会与|||左边的运算对象的后续情况有关，因而没有非确定性的问题。左边的运算会停止动作，而 b 与 c 之间的选择是留给环境

[191] 不要求两个进程之间的互相 lockup，互相耦合。

[192] 如果 x 和 y 是同一个事件，P 和 Q 是非确定性的被执行，不知道环境或者系统选择了哪一个。否则，如果执行了 x，剩下的 P 需要与系统的气体组成部分，例如，(y → Q)，再次形成 interleaving 的交叉关系。下次环境触发哪个事件都有可能。

全权处理。而在公式串的最右端公式中，由于选择是非确定性的，事件 a 可能是|||的两个运算对象中任意一个的一个事件。这样就无法由环境选择下一事件，使其为 b 或 c 了。

L6 和 L7 陈述了在|||的运算对象提供的初始事件中做出选择的是进程的环境，而不是其它的因素。而且只有当被选中的事件对两个运算对象均为可能事件时，才会出现非确定性问题。

例子

X1 设 R = (a → b → R)

那么 (R ||| R)

= (a → ((b → R) ||| R) □ a → (R ||| (b → R))) L6

= a → ((b → R) ||| R) ⊓ (R ||| (b → R)))

= a → ((b → R) ||| R) L2

而且(b → R) ||| R

= (a → ((b → R) ||| (b → R)) □ b → (R ||| R) L6

= (a → (b → ((b → R) ||| R)) 由上式

□ b → (a → ((b → R) ||| R)))

= μX • (a → b → X □ b → a → X) 卫式递归

由此可见，(R ||| 3.5 节 X2 很类似[193]。同理可证得

(VMS ||| VMS) = VMS2 □

3.6.2 迹和拒绝集(Traces and refusals)

(P ||| Q)的迹是由 P 和 Q 的迹的任意交叉构成的。交叉的定义请参看 1.9.3 节。

L1 traces(P ||| Q) = { s | ∃ t : traces(P) • ∃ u : traces(Q) • s interleaves (t, u) }

(P ||| Q)能执行对 P 或 Q 为可能的任何初始事件；因而它只能拒绝执行进程 P 和 Q 都拒绝的事件

L2 refusals(P ||| Q) = refusals(P □ Q)

执行完迹 s 中的事件后，(P ||| Q)的行为可由以下相当复杂的公式定义

L3 (P ||| Q) /s = $\sqcap_{t,u \in T}$ (P/t) ||| (Q/u)

其中

[193] 最后的结论是：如果 R = (a → b → R)，那么(R||R) = a → μX • (a → b → X □ b → a → X)

$$T = \{ (t, u) \mid t \in traces(P) \wedge u \in traces(Q) \wedge s \text{ interleaves } (t, u) \}$$

这条法则反映了这样一个事实，即由于我们无法知道(P ||| Q)的迹 s 是通过怎样的方式由进程 P 和 Q 的迹交叉构成的；因而继 s 之后，(P ||| Q)的未来行为可由 P 和 Q 的迹构成 s 的所有可能交叉方式中的任一种所反映。至于选择哪一种，我们并不知道，也无法确定。

3.7 描述(Specifications)

在 3.4 节中我们已经看到引入进程的拒绝集市很有必要的。拒绝集是可被间接观察到的进程行为的一个重要方面。因此在对一进程进行描述时，我们不仅需要刻画该进程迹的特性，而且要刻画其拒绝集的性质。类似我们使用 tr 来表示进程的迹，这里我们引入变量 ref 表示进程的一个任意拒绝集。当 P 为一非确定性进程时

$$P \text{ sat } S(tr, ref)$$

的含义就修改为[194]

$$\forall tr, ref \bullet tr \in traces(P) \wedge ref \in refusals(P/tr) \Rightarrow S(tr, ref)$$[195]

例子

X1 当一台自动售货机已收到的硬币比它已送出的巧克力的数目更多的时候，顾客有权要求该售货机不能拒绝再给出一块巧克力。可以形式表示如下:

$$FAIR = (tr\downarrow choc < tr\downarrow coin \Rightarrow choc \notin ref)$$

有一点是不言而喻的，即这个被描述进程的每个迹 tr 和每个拒绝集 ref 任意时刻都应满足上述描述。 □

X2 当售货机所收硬币和所给出的巧克力数目相当时，售货机的主人自然要求它不能拒绝再收一枚硬币。可以形式表示如下:

$$PROFIT1 = (tr \downarrow choc = tr \downarrow coin \Rightarrow coin \notin ref)$$ □

X3 简单自动售货机应当满足下述复合描述

$$NEWVMSPEC = FAIR \wedge PROFIT1 \wedge (tr \downarrow choc \leq tr \downarrow coin)$$

简单 VMS 满足该描述。而且象 VMS2(1.1.3 节中 X6)那样的售货机也满足它，尽管 VMS2 可以连续接受几枚硬币，并能连续送出几块巧克力。 □

X4 如果愿意，可以给连续接受硬币的自动售货机设一个上限。

---

[194] 参阅 1.10.2 和 2.7 节通过一个进程满足一个迹的 P sat S(tr)形式方法对进程的功能描述。

[195] P sat S(tr, ref)可以理解为：P 的行为满足 tr，而且之后的行为，虽然不知道行为是什么，但不能是什么(ref)很清楚。

ATMOST2 = (tr↓coin - tr ↓choc ≤ 2) □

X5 还可以坚持让机器在顾客愿意送入硬币时，一次至少可连续接受两枚硬币，

ATLEAST2 = (tr↓ coin - tr ↓ choc < 2 ⇒ coin ∉ ref) □

X6 进程 STOP 拒绝它的字母表中的任何事件。下面描述的是字母表为 A 的永远不停止动作的进程

NONSTOP = (ref ≠ A)

如果 P sat NONSTOP，且有一环境允许 A 中所有事件发生，则 P 必须执行其中的一个事件。由于(参看上例 X3)

NEWVMSPEC ⇒ ref ≠ {coin, choc}

因而凡是满足 NEWVMSPEC 的进程都永远不会停止动作。 □

以上几个例子说明了把拒绝集 ref 引入进程的描述后，能够表述进程的许多非常微妙却又极其重要的性质；也许这些性质中最重要的性质是，进程不停机(X6)。为得到这些有益的代数性质所付的代价是证明的规则及证明过程要稍微复杂些。

我们还希望证明的一点是，进程不会发散。3.8 节中叙述了为什么一个发散的进程能做任何事情，也可以拒绝任何事情。因此，如果存在这样一个事件集合，进程不能拒绝响应，那么这个进程就不是发散的。我们可以把进程不发散的这个充分条件写成公式

NONDIV = (ref ≠ A)

幸运的是

NONSTOP = NONDIV

因此证明一个进程不发散，并不比证明无死锁更麻烦。

3.7.1 证明(Proofs)

在下面的证明规则中，为方便起见，描述时而写成 S，时而写成 S(tr),或写成 S(tr，ref)的形式。但我们必须清楚，在任何写法中 tr，ref 都可能是进程描述的自由变量。

由非确定性的定义可知，(P ⊓ Q)或似 P 动作，或似 Q 动作。于是对其行为的每次观察就等于是对 P 或 Q 或二者兼有的观察。因而这种观察就可由 P 或 Q 或这两者的描述进行刻画。所以非确定性的证明规则的形式非常简单。

L1 如果 P sat S

而且 Q sat T

那么(P ⊓ Q) sat (S ⋁ T)[196]

STOP 的证明规则表达了该进程不做任何事情，而且拒绝执行任何事件

L2A $STOP_A$ sat (tr = <> ⋀ ref ⊆ A)

因为进程的拒绝集总是包含于它的字母表中(3.4.1 节 L8)，ref ⊆ A 其实可以省略不写。这样一来，如果我们把字母表一同略去(我们以后要这样做)，法则 L2A 就跟有关确定性进程的那条法则(1.10.2 节 L4A)完全一样，即

STOP sat tr = <>

前面有关前缀运算的法则(1.10.2 节 L4B)虽然仍有效，但已不足以证明进程在执行其初始事件之前不能停止动作这一点了。因此该法则需要加强，需要提及在进程的初始状态下，也就是当 tr=<>时，它的初始动作是不能被拒绝的。

L2B 如果 P sat S(tr)

那么 (c → P) sat ((tr=< > ⋀ c ∉ ref) ⋁ (tr_0 = c ⋀ S(tr´)))[197]

对有关一般选择的法则(1.10.2 节 L4)也需要类似地加强

L2 假设 ∀x ∈ B. P(x) sat S(tr，x)

那么(x：b → P(x)) sat

((tr = <> ⋀ (B ∩ ref) = {}) ⋁ (tr_0 ∈ B ⋀ S(tr´, tr_0))))

在 2.7 节中给出的有关并行组合的法则 L1 仍为有效的条件是，其描述不提及进程的拒绝集。为了正确处理进程的拒绝集，则需要一条更复杂一些的法则，即

L3 如果 P sat S(tr, ref)

且 Q sat T(tr, ref)

P 和 Q 均不发散

那么(P ‖ Q) sat

(∃ X,Y, ref • ref = (X ∪ Y) ⋀ S(tr ⍀ αP, X) ⋀ T(tr ⍀ αQ, Y))

改变符号的法则也需作类似修改

L4 如果 P sat S(tr，ref)

假如 f 为 1-1 函数, 则 f(P) sat S(f^{-1*}(tr), f^{-1}(ref))。

有关□的法则是出奇地简单

---

[196] (P ⊓ Q)满足 S 或者 T 的规约。

[197] tr´的 tr 属于(c → P)的迹，因此 tr´属于 P 的迹满足 S(tr´)。

L5 如果 P sat S(tr, ref)

且 Q sat T(tr, ref)

P 和 Q 都不发散

则(P □ Q) sat (if tr = <> then (S ∧ T) else (S ∨ T))

最开始的时候，即 tr=<>，(P □ Q)所拒绝的集合必须是 P 和 Q 都拒绝的。所以，该集合应由它们两者的描述所规定。此后，即当 tr≠<>，对(P □ Q)的观察是对 P 或 Q 的一个观察，因此必须被它们两者之一(或两者一起)的描述所规定。

关于交叉的法则则不必涉及拒绝集

L6 如 P sat S(tr)

且 Q sat T(tr)

P 和 Q 都不发散

则(P|||Q) sat (∃ s, t • (tr interleaves(s, t) ∧ S(s) ∧ T(t)))

有关屏蔽运算的法则，由于要防止进程的发散性，因而变得复杂了

L7 如果 P sat (NODIV ∧ S(tr, ref))

那么 (P \ C) sat ∃ s • tr = s ↾ (αP -C) ∧ S(s, ref ∪ C)

其中 NODIV 表示被屏蔽符号出现的次数不超过非屏蔽符号出现次数的某个函数值，即

NODIV = #(tr ↾ C) ≤ f(tr ↾ (αP- C))

这里的 f 是一个从进程的迹到自然数的全函数。

我们解释一下法则 L7 结论中的子句 ref∪C。由于仅当 P 能拒绝集合 X∪C 时，即拒绝 X 与所有被隐藏事件的全体时，P \ C 才能拒绝集合 X。因此只有当 P \ C 处于无法执行被屏蔽的内部动作的状态下，才不能拒绝与其外部环境发生相互作用。正如我们在 3.5.1 节中所述，这种公平性是屏蔽运算的任何合理定义的一个很重要的特征。

有关递归式(1.10.2 节 L6)的证明方法也需要强化。设 S(n)是含有变量 n 的一个谓词，n 的取值范围是自然数 0,1,2，...

L8 如果 S(0)

并且(X sat S(n)) ⇒ (F(X) sat S(n+1))

那么(μX.F(X)) sat (∀n • S(n))

这条法则甚至对非卫式的递归式也成立，不过所能证明的，使这类进程满足的最强的描述，是无意义的描述，即永真。

3.8 发散性(Divergence)

在前几章里，我们一直遵守着一个约定，即定义进程时，所用的方程式必须是卫式(1.1.2 节)[198]。这一限制不仅保证了方程式有唯一解(1.3 节 L2)，而且为我们免去了一个麻烦，不必给以下无穷递归式赋以含义

$\mu X \bullet X$

但不幸的是，引入屏蔽运算(3.5 节)后，一个明显的卫式递归式就不一定是构造性的了[199]。例如，考虑方程

$X = c \rightarrow (X \backslash \{c\})_{+\{c\}}$

$(c \rightarrow STOP)$和 $(c \rightarrow a \rightarrow STOP)$都是该方程的解；只要把它们代入方程就能检验出来，它们确实是解[200]。

因此，凡是涉及在屏蔽算子下有递归成分的那种递归方程，都是潜在的非卫式定义，而且其解可能不止一个。那么哪个解才是所求的解呢？我们约定最不确定的那个解是所求的解，因为这样就允许在所有其它的解中做非确定性的选择。这样一来，我们不仅能把递归式必须是卫式的这个限制条件甩掉，而且可以给形式为 $\mu X \bullet F(X)$的每个表达式一个(可能是非确定的)含义，表达式中的 F 可以使用本书中引入的任何算子(但/除外)，而且服从有关字母表的一切约束条件。

为了解释上述的含义，我们从最简单的情形入手。最简单的情形往往也是最坏的情形，即无穷递归式[201]。

$\mu X \bullet X$

我们知道，每个进程都是递归方程

$X = X$

的解，因此 $\mu X \bullet X$ 可以是任何进程般动作。这样它就成了所有进程中最不确定的，最难预测，也是最不容易控制的进程，简而言之，它是最糟的进程，让我们给它一个恰如其分的名字，称作浑沌且定义

$CHAOS_A = \mu X: A \bullet X$

---

[198] 在 Hoare 的 1985 年的原著里，强调了递归方程。"the quations which define a process by recursion must be guarded"。在 2015 年的新版本中，去除了"by recursion"。

[199] 请参阅 2.8.3 关于构造性的定义：$F(X) \restriction (n+1) = F(X \restriction n) \ (n+1)$ for all X。

[200] 存在多个解了。失去递归方程的唯一解的性质了。

[201] 在 CSP 2015 年的 online 版本 PDF 电子书中，关于 CHAOS 进程的分析和定义这一大段落被删除了。令人费解。相关段落可参阅 Hoare 的 1985 出版的 CSP 书中的 126 页和周巢尘的译作的 110 页。

我们前面给出的递归式的情形稍好些

$$\mu X \bullet (c \to (X \setminus \{c\})_{+\{c\}} = c \to cHAOS$$

这个进程与 CHAOS 是不同的，因为在它崩溃到 CHAOS 状态之前，它至少还可以执行初始事件 c。

进程 CHAOS 除了给无穷递归式赋予一个含义之外，还可以表示一个进程连续执行屏蔽事件构成的无穷序列造成的结局。最简单也是最糟的情形，就是我们前面在 3.5.1 节结尾引述的立刻发散进程。

$$(\mu X: A \bullet (c \to X) \setminus \{c\} = \mu X: (A-\{c\}) \bullet (c \to X) \setminus \{c\}$$

$$= \mu X:(A-\{c\}) \bullet (X \setminus \{c\}) \qquad \text{3.5.1 节 L5}$$

$$= \mu X:(A-\{c\}) \bullet X$$

$$= CHAOS_{A-\{c\}} \qquad \text{CHAOS 定义}$$

3.8.1 法则(Laws)

由于 CHAOS 是最不确定的进程，因而附加更多的非确定性选择，也不能改变它；于是它就成了⊓的零元素，即有

L1 $P \sqcap CHAOS = CHAOS$

我们称一个进程函数是严格的，如果只要其自变量中有一个是 CHAOS 时，整个函数值就会是 CHAOS。以上法则(再加上对称性)说明了⊓是严格的。进程 CHAOS 实在是太糟了，以致于凡用 CHAOS 定义的进程都等于 CHAOS。

L2 一下各个运算都是严格的

$$/\ s, \|, f, \Box, \setminus C, |||, \mu X$$

但是前缀运算不是严格的，因为

L3 $CHAOS \neq (a \to cHAOS)$

等式不成立的原因是，在右端变得完全不可靠之前，还可以指望它执行的事件 a。

我们前面提到过，CHAOS 是最难预测、最不容易控制的进程。因此没有它不能做的事；另外，也没有任何它不能拒绝执行的事件！即有如下法则

L4 $traces(CHAOS_A)=A^*$

L5 $refusals(CHAOS_A)=A$ 的全部子集[202]。

[202] 可以接受任何事件其实意味着可以拒绝任何事件。

3.8.2 发散集(Divergences)

如果一个进程执行完它的某一个迹之后，其行为进入浑沌状态，我们就定义这个迹为该进程一个发散迹。进程 P 的所有发散迹的集合定义如下

divergences(P) = {s | s ∈ traces(P) ∧ (P/s) = CHAOSαp}

由此定义立刻可得

L1 divergences(P) ⊆ traces(P)

由于/t 是严格的，因此有

CHAOS / t = CHAOS

从而得出，一个进程的发散迹集(简称发散集)是延展封闭的，即

L2 s ∈ divergences(P) ∧ t ∈$(\alpha p)^*$ ⇒ (s ^ t) ∈ divergences(P)[203]

因为 $CHAOS_A$ 可能拒绝一个字母表的任何子集，所以我们有如下法则。

L3 s ∈ divergences(P) ∧ X ⊆ αP ⇒ X ∈ refusal(P/s)

以上给出的三条法则陈述了任何进程的发散集的一般性质。下面的法则解释复合进程的发散迹是怎样由各个组成部分的发散迹和正常迹来确定的。首先，进程 STOP 绝不会发散，即

L4 divergences(STOP) = {}

而另一个极端情形是，进程 CHAOS 的每一个迹都导致 CHAOS

L5 divergences($CHAOS_A$) = A^*

以选择算子来定义的进程，第一步并不发散。因此它的发散迹集要由完成第一步动作之后的进程来确定

L6 divergences(x: b → P(x)) = {<x> ^ s |

x ∈ B ∧ s ∈ divergences(P(x))}

P 的任何一个发散迹也是(P ⊓ Q)和(P □ Q)的发散迹

L7 divergences(P ⊓ Q) = divergences(P □ Q)

= divergences(P) ∪ divergences(Q)

由于‖是严格算子，(P ‖ Q)的发散迹可能以 P 和 Q 的非发散动作组成的迹开始，但这个迹会导致 P 或 Q 发散(或使两者都发散)，即

L8 divergences(P ‖ Q) =

{s ^ t | t ∈ (αP ∪ αQ)* ∧

[203] 按照 divergenes 的定义，如果一个 s ∈ divergences(P)， 那么(P/s) = CHAOSαp。

((s ⌈ αP ∈ divergences(P) ∧ s ⌈ αQ ∈ traces(Q)) ∨

(s ⌈ αP ∈ tracesP) ∧ s ⌈ αQ ∈ divergences(Q))}

同理可以解释算子|||的发散迹集

L9 divergences(P ||| Q) =

{u | ∃ s, t • u interleaves(s, t) ∧

((s ∈ divergences(P) ∧ t ∈ traces(Q)) ∨

(s ∈ traces(P) ∧ t ∈ divergences(P)))}

由屏蔽运算造成的进程的发散迹集合，包含原进程的发散迹，和新增加的，试图屏蔽一个无穷的符号序列所得到的新的发散迹，故有

L10 divergences(P \ C) =

{(s ⌈ α(P-C)) ^ t |

t ∈ (αP-C)* ∧

(s ∈ divergences(P) ∨

(∀ n • ∃ ∈ C* • #u＞n ∧(s^u) ∈ traces(P)))}

仅当原进程的迹发散时，经符号变换所得到的进程迹才会发散，故有

L11 divergences(*f*(P)) = {*f**(s) | s ∈ divergences(P)}

其中 *f* 是一一对应函数。

发散现象是我们所不希望的东西，可我们却花费这么大精力去讨论它，似乎很不应该。但不幸的是，发散现象的存在似乎是任何一个有效的进程实现方法中不可避免的一个现象。不论是屏蔽运算，还是非卫式递归式，都可以引起进程的发散；因而系统设计者的任务之一就是证明在他的具体设计中，这种问题不会发生。但是为了证明某件事不会发生，我们需要一种有关此事能否发生的数学理论！

3.9 非确定性进程的数学理论(Mathematical theory)

本章给出的法则比前两章中的那些法则要复杂得多；从而导致本章中这些法则的非形式证明及有关例子也相应地令人难以信服。所以，构造非确定性进程概念的恰当的数学定义，以及依据各个算子的定义严格证明给出法则的正确性就显得更为重要。

我们在 2.8.1 节中讲过[204]，数学模型是以进程的可直接或间接观察到的有关性质为基础的。这些性质当然包括进程的迹和字母表在内；但是对非确定性进程来说，它的拒绝集(3.4 节)和发散集(3.8 节)也属于这类性质[205]。而且在除了进程 P 初启时的拒绝集之外，我们还需要考虑到当它执行完其行为的一个任意迹 s 之后，它的可能的拒绝集。为此我们定义进程的失败集，并表示为如下的一种二元关系。

failures(P) = { (s, X) | s ∈ traces(P) ∧ X ∈ refusals(P / s) }

如果(s, X)是进程 P 的一个失败，它的含意是，P 可以执行由迹 s 记录的事件序列，然后就拒绝再做更多的事情，尽管环境已做好执行 X 中任何事件的准备[206]。进程的失败集与其正常的迹集合和拒绝集相比，更能说明一个进程的行为，而且后两者均可用失败集来定义

traces(P) ={ s | ∃ X • (s, X) ∈ failures(P) }

= domain(failures(P))

refusals(P) = {X | (<>, X) ∈ failures(P) }

因此，进程的正常迹集的各种性质(1.8.1 节 L6, L7, L8)，以及拒绝集的各种性质(3.4.1 节 L8, L9, L10, L11)就很容易地用失败集重新形式表示(参看下面定义 D0 中的条件 C0, C1, C2, C3)。

现在我们可以大胆地断论：一个进程是由描述其字母表、失败集以及发散集，这三个集合唯一地确定的；反过来说，任何三个这样的集合，只要满足有关的条件，可以唯一地确定了一个进程。下面我们给出一些新定义之前，首先先定义 A 的幂集，幂集是 A 的所有子集的集合。令

ℙA={X | X ⊆A }

D0 一个进程是一个三元组(A, F, D)，其中

A 是符号的集合(为简单起见我们规定它为有限集)

F 是 A*和 ℙA 之间的某种对应关系

D 是 A*的一个子集

---

[204] 确定性进程的数学模型

[205] 为什么需要引入拒绝集来形式化描述非确定性进程，是因为，例如，对于，(P ⊓ Q)与(P □ Q)，不能根据它们的迹把它们区分开，因为它们中任一个的迹同时也可能是另一个进程的迹。但是如果把它们放到某个特定的环境中，在这个环境中，(P ⊓ Q)一开始就可能死锁，而(P □ Q)没有死锁，例如如果 P 和 Q 的初始事件完全不一样。简单的理解可以是：即使环境给了 P 能够接收的事件，但(P ⊓ Q)可能选择 Q，从而导致 STOP，进入一个拒绝状态。

[206] 更准确的定义应该理解为：是可能不再响应环境的事件。但不一定真的发生。但存在这个可能。例如，注 201 所示。

它们满足以下条件

C1 (<>, {}) ∈ F

C2 ($s^\frown t$，X) ∈ F ⇒ (s,{}) ∈ F

C3 (s, Y) ∈ F ∧ X ⊆ Y ⇒ (s, X) ∈ F

C4 (s, X) ∈ F ∧ x ∈ A ⇒ (s, X ∪ {x}) ∈ F ∨ ($s^\frown$<x>, {}) ∈ F

C5 D ⊆ domain(F)

C6 S ∈ D ∧ t ∈ A* ⇒ $s^\frown t$ ∈ D

C7 s ∈ D ∧ X ⊆ A ⇒ (s, X) ∈ F

(后三个条件即 3.8.2 节中法则 L1，L2，L3)。

满足如下定义的最简单的进程就是那个最糟的进程

D1 $CHAOS_A$ = (A, (A^*×ℙA), A^*)

其中 A^*×ℙA 是笛卡尔积

{(s, X) | s∈ A* ∧ X ∈ ℙA}

该进程是所有字母表为 A 的进程中最大的，因为 A^*的每个元素既是进程的一个正常迹，也是它的一个发散迹，而且 A 的每个子集，都是继进程的任意迹之后的一个拒绝集。

另一个简单的进程是

D2 $STOP_A$ = (A,{< >}×ℙA,{})

这个进程从不动作，而且拒绝一切，也不发散。

如果需要定义一个进程算子，需要说明结果进程的那三个集合是怎样由算子的运算对象的三个集合演变而来的。当然还有必要说明这个结果进程也满足 D0 的六个条件；有关这一点的证明是建立在这样一个假设的基础之上的，即构造结果进程依据的算子的运算对象是满足那六个条件的。

要定义的最简单的进程运算是非确定性的或运算(⊓)。象许多其它算子一样，它的定义只对具有相同字母表的运算对象适用。其定义为

D3 (A, F1, D1) ⊓ (A, F2, D2) = (A, F1 ∪ F2, D1 ∪ D2)

该算子的两个运算对象中的任何一个的可能失败或发散迹，都是结果进程的可能失败或发散迹。3.2.1 节中的法则 L1, L2,L3 是本定义的直接推论。

类似地可以给出所有其它算子的定义；可是如果能写出进程的字母表、失败集及发散迹的各自的定义，似乎就更优美些。3.8.2 节中我们已经给出了发散迹的定义，因此剩下的工作就只是定义字母表和失败集。

D4 如果 $\alpha P(x) = A$ (对一切 x)

且 $B \subseteq A$

那么 $\alpha(x : B \rightarrow P(x)) = A$.

D5 $\alpha(P \parallel Q) = (\alpha P \cup \alpha Q)$

D6 $\alpha(f(P)) = f(\alpha(P))$

D7 $\alpha(P \Box Q) = \alpha(P ||| Q) = \alpha P$ 假设 $\alpha P = \alpha Q$

D8 $\alpha(P \setminus C) = \alpha P - C$

D9 failures$(x: B \rightarrow P(x)) =$

$\{<>, X \mid X \subseteq (\alpha P - B)\} \cup$

$\{ (x)^\frown s, X \mid x \in B \wedge (s, X) \in \text{failures}(P(x)) \}$

D10 failures$(P \parallel Q) =$

$\{ s, (X \cup Y) \mid s \in (\alpha P \cup \alpha Q)^* \wedge (s \upharpoonright \alpha P, X \in \text{failures}(P) \wedge$

$(s \upharpoonright \alpha Q, Y) \in \text{failures}(Q) \} \cup$

$\{s, X \mid s \in \text{divergences}(P \parallel Q) \}$

D11 failures$(f(P)) = \{ f^*(s), f(X) \mid (s, X) \in \text{failures}(P) \}$

D12 failures$(P \Box Q) =$

$x\{ s, X \mid (s, X) \in \text{failures}(P) \cap \text{failures}(Q) \vee$

$(s \neq <> \wedge (s, X) \in \text{failures}(P) \cup \text{failures}(Q)) \} \cup$

$\{ s, X \mid s \in \text{divergences}(P \Box Q) \}$

D13 failures$(P ||| Q) =$

$\{ s, X \mid \exists\ t, u \bullet (t, X) \in \text{failures}(P) \wedge (u, X) \in \text{failures}(Q) \}$[207] $\cup$

$\{s, X \mid s \in \text{divergences}(P ||| Q) \}$

D14 failures$(P \setminus C) =$

$\{s \upharpoonright (\alpha P - C), X \mid (s, X \cup C) \in \text{failures}(P) \} \cup$

$\{ s, X \mid s \in \text{divergences}(P \setminus C) \}$

[207] 在 1985 年的 CSP 书里，D13 需要指明 s interleaves(t,u)。2015 年的 CSP 在线电子书移除了这一点，可能是认为不需要特别指明。

在解释以上这些法则时，可以从解释其相应的迹和拒绝集入手，并借助于算子 / 的有关法则。

我们剩下的工作是给出用 μ 算子递归定义的进程定义。整个处理方法基于 2.8.2 节中的不动点理论，只是次序关系⊆的定义有所不同

D15 $(A, F1, D1) \subseteq (A, F2, D2) \equiv (F2 \subseteq F1 \wedge D2 \subseteq D1)$[208]

P ⊆ Q 说明 Q 与 P 相等或 Q 比 P 更好[209]。这可以理解为，相比之下 Q 更不容易发散，也不易陷于失败。Q 比 P 更易预测，更易控制，因为 Q 可能做的不该做的事情，P 也同样可以做；另外，Q 如果能拒绝执行某件该做的事情，P 也一定能够拒绝这个事情。CHAOS 可以做任何字母表中的事件，也可以拒绝任何字母表中的事件，不受任何限制。因此，它是所有进程中名副其实地最不可预测的不可控制的进程，或简单地说，它是最糟的进程。因此有

L1 $CHAOS \subseteq P$

这个次序关系很显然是一种偏序关系。事实上它是个完全偏序，其序列的极限是以递降的失败集和发散集的交集定义的

D16 如果$(\forall n \geq 0 \bullet F_{n+1} \subseteq F_n \wedge D_{n+1} \subseteq D_n)$

$$\bigsqcup_{n\geq 0} (A, F_n, D_n) = (A, \cap_{n\geq 0} F_n, \cap_{n\geq 0} D_n)$$[210]

μ 算子的定义方法与确定性进程的定义方法(2.8.2 节 L7)相同，区别仅在于对次序关系的定义上。在现在的次序关系下，要以 CHAOS 代替 STOP，即

D17 $\mu X: A \bullet F(x) = \bigsqcup_{i\geq 0} F^i(CHAOS_A)$

有关上式为某个相关方程的解(其实是最不确定的解)的证明与我们在 2.8.2 节中给出的那个证明相同。

与以前相关章节一样，这个证明能否有效的关键是递归式右端使用的所有算子在适当的次序关系下是否连续的。幸运的是本书中定义的所有算子(/除外)都是连续的，因而以这些算子构成的表达式也都是连续的。本书中对发散的复杂处理的一个主要原因，就在于保证屏蔽算子/的连续性。

---

[208] 与 2.8.2 的确定性进程的数学模型一节不同的是，非确定性进程的次序关系是通过其失败集，发散集的关系来刻画。单纯的用正常的行为迹集合不能够完善的表达进程之间的关系了。例如，2.8.2 的 D1 定义：**D1** $(A, S) \subseteq (B, T) = (A = B \wedge S \subseteq T)$

[209] Q 比 P 更好的意思可以理解为：P 能做的 Q 都能做。Q 比 P 更加”正常“。

[210] $\bigsqcup_{n\geq 0} (A, F_n, D_n)$是失败和发散最小的下限。或者说，是能做事情做多的那个进程。这样理解与在 2.8.2 中确定性进程数学模型中的定义就可以一致性的理解了。

第四章　通信(COMMUNICATION)

4.1 引言(Introduction)

在前几章里，我们介绍了广义的事件(event)概念。事件是一个瞬间，无延时的动作，事件动作的发生可能需要一个以上的独立刻画的进程同时参与。本章中，我们将着重讲述一类称之为“通信”的特殊事件。所谓通信就是用二元对 c .v 描述的一个事件，其中 c 是发生通信的通道的名字，v 是通信时在通道 c 上传递的消息。我们在 1.1.3 节 X7 的 COPYBIT 和 2.6 节 X4 的 $CHAIN_2$ 中其实已经给出了这种约定记法的两个例子。我们定义进程 P 能在通道 c 上传递的所有消息的集合为

$$\alpha c(P) = \{ v \mid c.v \in \alpha P \}$$

我们还定义如下两个函数，它们可将通信的通道和消息分离出来

$$channel(c.v) = c,\ message(c.v) = v$$

在这一章里引入的所有操作都可以用前几章里的那些更基本的概念来定义，而且大部分的法则也只是前面大家所熟悉的那些法则的特例。为什么还要引入这些特殊的符号是因为可以使我们巧妙的运用到一些很有用的应用和实现方法上；而且在某些情况下，对这些施加了限制的符号可运用更为有力的推理方法[211]。

4.2 输入和输出(Input and Output)

设 v 是 αc(P)中的一个元素。一个进程，如果先在通道 c 上输出 v, 然后执行进程 P 的动作，可以定义为

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

这个进程初启准备执行的唯一的事件就是通信事件 c.v。

如果一个进程最初准备在通道 c 上输入可传递的任意值 x,然后执行 P(x)动作，我们就将它定义为

$$(c?x \rightarrow P(x)) = (y : \{y \mid channel(y) = c\} \rightarrow P(message(y)))$$[212]

例子

X1 应用输入和输出的新定义，我们可以把 1.1.3 节中 X7 改成为[213]

---

[211] 可以理解为通信事件的定义在概念上比较能够形象的表达两个进程的生产者和消费者的关系。是对普通的一般性的事件概念的一个语义上的封装，用来描述两个行为体的触发依赖关系。

[212] P(message(y)就是 P 收到通道 c 上的消息 y 之后的相应行为。

COPYBIT=μX • (in?x → (out!x → X))

其中 αin(COPYBIT) = αout(COPYBIT) = {0, 1} □

我们将遵守这样一个约定，即每条通道仅用于单向通信，而且仅用于发生在两个进程之间的通信。为某进程输出信息的通道叫作该进程的输出通道；只为某进程输入信息的通道叫作该进程的输入通道。进程的输入通道或输出通道的名字都是(不很严格地)该进程字母表中的元素。

当画一个进程的连接框图时(2.4 节)，通道被画成是指向适当方向的箭头，并用通道名做上标记(见图 4.1)。

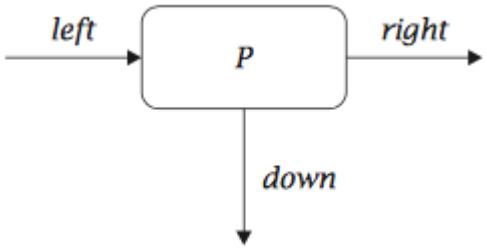


图 4.1

假设 P 和 Q 为进程，c 为 P 的一个输出通道，同时是 Q 的输入通道。当 P 和 Q 并发组合成系统(P ‖ Q)时，则每当 P 在通道 c 上输出一个消息的值，输入进程 Q 同时会接受这个相应的消息。一个输出进程可以指定消息的具体的值，但输入进程只能准备好接受可传递的任意值。因此实际将要发生的事件如果是通信 c.v，其中的 v 就是由输出进程所确定的那个值。显然，一个明显的约束条件是通道 c 的两端的字母表必须相同，即

αc(P) = αc(Q)

以后我们假设上述约束条件总是成立的；而且在不引起混淆的情况下，我们将把 αc(P) 和 αc(Q)简记为 αc。其实 2.6 节 X4 中的 CHAIN2 就是上述通信模型工作原理的一个实例；在 4.3 节及其后的几节里我们还要给出几个更有意思的例子。

一般说来，进程要输出的值是由一个表达式确定的，其所含的变量的值是由前面某个输入操作所赋予的，以下几个例子会阐述相关机制。

例子

X1 一进程把它从左端输入(收到)的每个消息立即拷贝下来并输出到右端去，该进程记为

[213] 第一章中基于简单事件的定义为 COPYBIT = μX • (in.0 → out.0 → X | in.1 → out.1 → X)。显然，通过通信事件的刻画，COPYBIT 进程的定义更加紧凑，语义表达的更加清晰。

αleft(COPY) = αright(COPY)

COPY = μX • (left?x → right!x → X)

如果 αleft ={0, 1}，则 COPY 就与 COPYBIT(1.1.3 节 X7)几乎完全一样了。 □

X2 一个进程与 COPY 类似，只是每个输入的数字在输出前都增大一倍，则有

αleft = αright = ℕ

DOUBLE=μX • (left?x → right!(x+x) → X) □

X3 一张穿孔卡上记载的值是 80 个字符构成的一个序列，整张卡片可当作单个值沿左通道传入。那么负责读入卡片，然后将其上的字符一个一个地输出的进程可记为

αleft = {s | s ∈ αright* ∧ #s = 80}

UNPACK = $P_{<>}$

其中 $P_{<>}$ = left?s → P_s

$P_{<x>}$ = right!x → $P_{<>}$

$P_{<x>^s}$ = right!x → P_s [214] □

X4 一个进程由左通道一个一个地输入字符，然后将它们组合成 125 个字符长度的字符行。每组合成一行，就把该行当作一个具有数组值的消息，由右通道输出，整个进程记为

αright = { s| s ∈ αleft* ∧ #s = 125}

PACK =$P_{<>}$

此处 P_s = right!s → $P_{<>}$ 若#s=125

P_s = left?x → $P_{s^{<x>}}$ 若#s＜125

这里，P_s所描述的是进程从输入通道接收符号并将其排列成字符串 s 后的行为；要等到该行足够长之后所有字符才被输出。 □

X5 一个进程从左至右拷贝消息，但把连续输入的一对星号用一个“↑”来代替

αleft = αright -{"↑"}

SQUASH =μX • left?x →

if x ≠ “*” then (right!x → X)

else left?y → (if y = “*”then (right!“↑” → X)[215]

else (right!"*" → right!y → X))[216] □

[214] 在右边的通道输出目前的第一个字符<x>，然后接着处理后续的字符串 s。

[215] 遇到了连续的两个*， 选择在右侧的通道输出↑。

[216]只是一个单个的*，选择直接输出第一个*和 y。

考虑这样的进程，一开始在一组通道中的任意一个通道上通信，等待输入事件的触发，至于会是哪个通道留给与这个进程相连接的其它进程去决定或者触发。我们采用在第一章中引入的选择(Choice)记号。设 c 和 d 为不同的通道名，那么

(c?x → P (x) | d?y → Q(y))

所表示的进程，它一开始或从 c 上接收 x，然后执行 P(x)动作，或从 d 上接收 y，然后执行 Q(y)动作。选择是由各通道上相应的输出方准备就绪的先后所确定的[217]，对此我们解释如下。

由于我们不考虑事件的时间以及执行事件的进程的速度，那么上一段的最后一句话就要解释一下[218]。我们先来考虑，当通道 c 和 d 分别是两个独立的进程的输出通道时的情形。我们说这两个进程是独立的，意味着它们既不直接也不间接地互相通信。因此这两个进程的动作是任意交互发生的。这样如果一进程正在为在通道 c 上实现一输出动作而努力，而另一进程又在为在 d 上实现输出动作而努力的话，决定不了它们中哪个首先完成输出[219]。这样的模型的实现者在解决这种非确定性时，需要并且一般倾向于选择先做好输出准备的那个进程，但这一点是不能强制每个实现都遵从的。这种策略也能防止死锁。例如，当第二个输出永远也发生不了，或它的发生只能后于第一个输出，比如通道 c 和 d 都被连接到同一与其并发的进程上，这个进程先在一个通道上输出，然后再在另一个通道上输出。即有

(c!2 → d!4 → P)

在这种情况下，如果不采用上述策略，就可能出现死锁。

因此，我们可以认识到，使用几个输入通过选择(Choice)的方式不仅可防止死锁发生，而且能提高通信的效率，减少对外界建议通信的答应时间。比方说，有位乘客在等 127 路公共汽车。一般说来，他等待的时间就要比即可乘 19 路，也可乘 127 路车的人等的长，因为不论哪趟车先到，后者都能上。假设车的到站是随机的，那么那位有选择余地的乘客等车的时间只需另一位的一半——换句荒唐点的话，好像他上两次车，另一位只上一次！采用等待许多可能事件中的第一个事件，是达到这类高效目标的唯一途径，只靠买一台运算速度更快的计算机是没有用的[220]。

[217] 可以理解为 c 通道或者 d 通道上哪个先有数据，就先触发相应的事件和后续动作。
[218] 指如何确定不同通道上的事件的先后问题。
[219] 因为在理论模型里，我们不考虑事件产生的时间因素。
[220] 也可以用排队轮里的观点来理解，多个队列一定比单个队列强。更不容易产生队列堵塞和拥挤。

X6 进程接受由两个通道 left1 和 left2 中的任意一个上的输入，并将消息立即由右边通道输出去，该进程可记为

αleftl = αleft2 = αright

MERGE = (leftl?x → Right!x → MERGE | left2?x → right!x → MERGE)

这个进程输出的是它从 left1 和 left2 输入的消息的交叉合并起来的数据。 □

X7 时刻准备着从左通道接收一个值，或把它刚刚接收的值输出到右通道的进程可记为

αleft = αright

VAR = left?x →VAR_x

其中 VAR_x =(left?y → VAR_y | right!x → VAR_x)

VAR_x的行为就象是当前值为 x 的程序变量，左通道上的通信给它赋新值，右通道上的通信负责取出其当前值。如果 αleft{0,1}，那么 VAR 的行为就与 BOOL(2.6 节 X5)的行为几乎完全一样。 □

X8 一个进程从 up 和 left 通道输入数据，并向 down 通道输出输入值的某个函数值，然后重复动作

NODE(v) = μX • (up?sum → left?prod → down!(sum + v * prod) → X)

□

X9 进程时刻准备从左通道输入消息，并把已输入的但尚未输出的每一个消息输出到右通道，这个进程写做

BUFFER = $P_{\langle\rangle}$

其中

$P_{\langle\rangle}$ = left?x → $P_{\langle x\rangle}$

$P_{\langle x\rangle^\frown s}$ = (left?y → $P_{\langle x\rangle^\frown s^\frown y}$ | right!x → P_s)

BUFFER 的行为就是个队列，消息由队列右端入队，再由左端离队，离队次序与它们加入时的次序相同，但消息由进至出要经过一段可能的延迟，在这段时间里，后来的消息还可以加入队列。 □

X10 有这样一个进程，其行为象个消息栈，当它是空时，它响应 empty 信号。在任何时刻它都准备好从左通道输入新的消息，并把它放在栈顶；只要不空，它也准备把栈顶的元素移出，即

STACK = $P_{<>}$

其中 $P_{<>}$ = (empty → $P_{<>}$ | left?x → $P_{<x>}$)

而且 $P_{<x>^\frown s}$ = (right!x → P_s | left?y → $P_{<y>^\frown<x>^\frown s}$)

这个进程与前例中的进程很相似，所不同的是，当它为空时，它还参予 empty 事件，而且它把刚送到的消息放在贮存序列中输出消息的一端。譬如说，y 是刚到的消息，x 是它正准备要输出的消息，那么 STACK 就贮存消息序列<y>^<x>^s，而 BUFFER 贮存的序列却是<x>^s^<y>。 □

4.2.1 实现(Implementation)

在通信进程的 LISP 实现中，事件 c.v 很自然的可用打了标点的二元对(c.v)来表达，其结构是

cons("c.v)

我们把输入和输出指令当作是以通道名为自变量的函数，这样，实现时就方便多了。如果进程没有准备好在该通道上的通信，它就回答"Bleep。至于被传递的实际数值，将在下一阶段分开处理，可看如下说明。

假设 Q 是输入命令

c?x → Q(x)

那么 Q("c) ≠ BLEEP; Q("c)的结果是以输入值 x 为其自变量的一个函数，且函数结果为进程 Q(x)。因此调用如下 LISP 函数就可实现输入命令 Q

Input("c, λx • Q(x))

这个函数的定义是

input(c, F) = λy • if y ≠ c then "BLEEP else F

由此可见，当<c.v>是 Q 的一个迹时，在 LISP 语言中，Q/<c.v>可通过 Q("c)(v) 来表达。

假设 P 是输出命令

c!v → P′

那么 P("c) ≠ BLEEP，而它的结果是二元对 cons(v, P′)。因此调用 LISP 函数

output("c, v, P′)

就可实现输出命令，这个函数的定义是

output(c, v, P) = λy • if y≠c then "BLEEP else cons(v, P)

显然可见，当<c.v>是 P 的一个迹时，在 LISP 语言里，v = car(P("c)), P/<c.v>可用 cdr(P("c))来表示。

在理论上，如果 αc 是有限集合，可以把 c.v 当作一个单一事件来处理，当作输入和输出命令的一个参数来传递。不过这么做的效率会低得可怕，为了找到输出的

值，需要对 αc 中的所有的值 v，逐个检验 P(c.v) ≠ "BLEEP 是否成立，直至找到了匹配的那个值为止。我们引入有关输入和输出的特殊记号的理由之一，就是为了鼓励使用高效的实现方法。这样做的缺点是，几乎所有其它算子也必须按照优化的要求重新实现。

例子

X1 COPY = LABEL X • input("left, λx • output("right, x,X)) □

X2 PACK = P(NIL)

其中

```
P = LABEL X.
    λs •  if length(s) = 125 then
              output( "right, s, X (NIL))
          else
              input( "left,
                  λx • X (append(s, cons(x, NIL))))
```

□

4.2.2 规约(Specifications)

在描述一个通信进程的行为时，简便的方法是，分别刻画各通道上传递的消息序列。假设 c 为一通道名，我们定义(参看 1.9.6 节)

tr↓c = message* (tr ↾ αc)

为方便起见，干脆把 tr↓省去，将 tr↓right ≤ tr↓left 简记为 right ≤ left。

另外一个非常有用的，来限定前缀长度的下限的定义是

$s \leq^n t = (s \leq t \wedge \#t \leq \#s + N)$

上式的含意是 s 为 t 的一个前缀，且 t 中删去的项最多不超过 n 个。以下衍生的法则都是显而易见而且相当有用的，如

$s \leq^0 t \equiv (s = t)$

$s \leq^n t \wedge t \leq^m u \Rightarrow s \leq^{n+m} u$

$s \leq t \equiv \exists\ n \bullet s \leq^n t$

例子

X1 COPY sat right $\leq^1$ left □

X2 DOUBLE sat right $\leq^1$ double* (left) □

X3 UNPACK sat right $\leq$ ^/ left

其中$^/<s_0, s_1, ..., S_{n-1}>= s_0 \,^ s_{1...} \,^ s_{n-1}$ (参看 1.9.2 节)

这说明右通道上的输出序列是，将左通道上输入的符号序列的串平铺成的一个序列。

□

X4 PACK sat (($^/$ right $\leq^{125}$ left) $\wedge$ (#*right $\in$ {125}*))

这个规约描述了右通道上输出的每个元素都是一个长度为 125 的序列，另外，所有右通道上的这些序列的合并来自于左通道上输入通道产生的消息序列的初始序列。 □

如果⊕是一个二元运算符，我们可以很方便的定义，将其扩充分配到两个序列的相应元素上。 我们定义结果序列的长度等于那个较短操作数的长度。即

$s \oplus t = \langle\rangle$ 如果 s = ⟨⟩ 或者 t = ⟨⟩

$=\langle s0 \oplus t0\rangle \,^\wedge\, (s' \oplus t')$ 如果 s 和 t 都不是空串

显然，

$(s \oplus t)[i] = s[i] \oplus t[i]$ 对于 $i \leq \min(\#s, \#t)$

而且

$s \leq^n t \Rightarrow (s \oplus u \leq^n t \oplus u) \wedge (u \oplus s \leq^n u \oplus t)$

X5 斐波那契序列

<1,1,2,3,5,8，... >是由如下递推关系定义的

fib[0] = fib[1] = 1

fib[i+2] = fib[i+1] + fib[i]

使用将序列左移一位的算子，即可得到上述第二式的另一种形式

fib'' = fib' + fib

我们把这个方程两边加上下标，可以从这个比较隐晦的形式里还原出斐波那契序列的原始定义来，由

fib'' [i] = (fib' +fib)[i]

于是 fib' [i + 1] = fib' [i] + fib[i] (1.9.4 节 L1)

故 fib[i + 2] = fib[i + 1] + fib[i]

有关这个方程式的另一种解释是，它是对无穷和的一个刻画，从下面的图中，可以明显的看出序列的左移：

1 , 1 , 2 , 3 , 5 ,... fib

↙ ↙ ↙ ↙

1 , 2 , 3 , 5 , ... +fib'

↙ ↙ ↙

2 , 3 , 5 , ... = fib"

在以上的讨论中，是把 fib 作为一个无穷序列来处理的。如果 s 是 fib 的一个有限初始子序列(且#s ≥ 2)，那么我们得到的就不是等式，而是一个不等式了

s" ≤ s' + s

这个公式可以用来描述，把斐波那契序列输出到右通道上的进程

FIB sat (right ≤ <1, 1> ∨ (<1, 1> ≤ right ∧ right" ≤ right' + right)) □

X6 值为 x 的一个变量在右通道上输出刚刚由左通道输入的值，如果没有输入的话，就输出 x。更为形式地叙述是，如果这最近的动作是个输出动作，那么被输出的值就等于序列<x>^left 的最后一项，即

VAR_x sat (channel($\overline{tr}_0$) = right ⇒ $\overline{right}_0$ = ($\overline{< x > \hat{}left}_0$)

其中$\overline{s}_0$是序列 s 的最后一个元素(参看 1.9.5 节)。

这个例子说明只用进程各个通道上的消息序列本身是不足以描述某些进程的。我们还需要知道它们各通道上通信事件交叉发生的次序，比如要知道最后的一次通信时是在右通道上发生的。一般说来，这种额外的复杂性对使用选择算子的进程是必不可少的。 □

X7 进程 MERG 输出的是，两个通道 left1 和 left2 上输入的两个序列的穿插序列(参看 1.9.3 节中定义)，并至多缓存一个消息，令

MERG sat ∃ r • right $\leq^1$ r ∧ r interleaves (leftl, left2) □

X8 BUFFER sat right ≤ left □

满足描述(right ≤ left)的进程刻画了一个透明的通信协议的行为，这个协议保证右通道输出的只是那些已从左通道输入的消息，而且不改变消息间的原来次序。不管消息输出的地方与它们被接受的地方相隔多远，也不管连接这两地之间的通信媒介是否一定可靠，通信协议必须保证满足上述描述。有关的例子我们将在 4.4.5 节中给出。

4.3 通信(Communications)

设 P 和 Q 为进程，c 为通信通道，P 用它输出，Q 用它输入。于是所有形式为 c.v 的通信事件的集合就包含于 P 和 Q 的字母表的交集之中。如果我们把这两个进程并发地组成系统(P ‖ Q)，则通信事件 c.v 只有当这两个进程同时执行它时才能发生，也就是说，只有当 P 在通道 c 上输出值 v，Q 又同时将它输入的时候，c.v 才能发生。由于

输入进程总是准备好接受任意可传递的值，因此，类似 2.6 节中 X4 所阐述的，每次通信中决定传递的实际消息值的是输出进程。

这样，输出事件可以被看作是前缀算子的特例，而输入则可以被看作是选择算子的特例；于是可以导出如下法则

L1 $(c!v \rightarrow P) \parallel (c?x \rightarrow Q(x)) = c!v \rightarrow (P \parallel Q(v))$

注意 c!v 是作为整个系统行为中的一个可观察到的动作保留在这个方程式的右边的。这相当于在连接系统各组成部分的连接线上分接出一条窃听线，从而能时刻记录系统组成部分之间的内部通信情况。另外，这还有助于整个系统的论证。

因为 c!v 是内部动作，如有必要，我们可将这些内部操作屏蔽起来，即在经同一通道相互通信的两个进程所构成的并行组合系统的外面运用屏蔽算子(参见 3.5 节)，这可表示为如下法则

L2 $((c!v \rightarrow P) \parallel (c?x \rightarrow Q(x)))\backslash C = (P \parallel Q(v))\backslash C$

其中 $C=\{c.v \mid v \in \alpha c\}$

有关的例子将在 4.4 节和 4.5 节中给出。

如果我们用通道名来表示在通道上传递的消息序列，那么多个通信进程组成的并行组成的组合系统的描述具有相当简单的形式。假设 c 是通道名，进程 P 和 Q 沿通道 c 通信。那么在 P 的描述里，c 表示 P 在通道 c 上传递的消息序列。同理，在 Q 的描述里，c 表示 Q 在通道 c 上传递的消息序列。

幸运的是，由通信的本性可知，当 P 和 Q 在 c 上通信时，它们送出或接收到的消息序列在任何时刻都必须是相同的。因此这个序列必须同时满足 P 和 Q 的描述。对 P 和 Q 字母表交集内所有通道来说，上述都是成立的。

现在考虑在 P 的字母表中，但是不在 Q 的字母表中的通道。由于这个通道不可能在 Q 的描述中被提到，所以在它上面传递的值就仅限于 P 的描述。同样道理，决定仅属于 Q 的通道上的通信的性质的是 Q 自己，于是只要简单地把 P 和 Q 的描述逻辑与起来，就可形成(P ‖ Q)的行为描述。显然，只有在用通信的通道名来表述这种组合结构时，P 和 Q 的描述才具有这种简洁性；值得注意的是，由 4.2.2 节 X6 可知有些情况下，只用通道名有时是不够的。

例子

X1 假设

$P = (left?x \rightarrow mid!(x \times x) \rightarrow P)$

Q = mid?y → right!(173 × y) → Q)

显然

P sat (mid $\leq^1$ square*(left))

Q sat (right $\leq^1$ 173 × mid)

这里(173 × mid)表示 173 乘以 mid 通道上每个消息。

这样可得到

(P ‖ Q) sat (right $\leq^1$ 173 x mid) ∧ (mid $\leq^1$ square* (left))

而这个描述蕴含

right ≤ 173 x square* (left)

这个公式正是我们原来想满足的描述。 □

用并发算子‖连接通信进程时，所得到的公式往往使人联想到，用通信线路连接电子部件的物理实现方法。这类实现的目的是为了加速产生所要的结果。特别是，当对输入的数据流的每一项都施行同样计算，而且又要求输出和输入的速率相同(一开始可能有延迟)时，这种技术很有效。实现这类数据加工的系统叫做数据流网络。

通信进程系统的示意图和它们的物理实现十分类似。一个进程的输出通道和另一个进程的同名的输入通道相连，只属于一个进程字母表的通道则处待连状态。图 4.2 是 X1 的示意图。

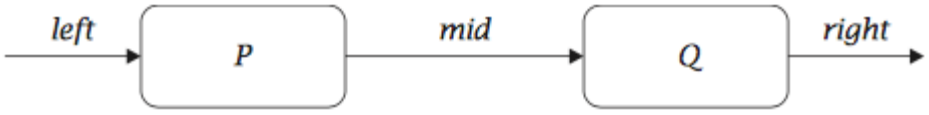


图 4.2

X2 两支数据流从通道 left1 和 left2 输入。由 left1 读入的 x 和从 left2 读入的 y 产生新数据(a×x+b×y)，并将产生的数据从右方的通道输出。为满足加工速度的需求，乘法必须并发运行。为此，我们定义两个进程，并把它们组合起来，即

X21 = (leftl?x → mid!(a×x) → X21)

X22 = (left2?y → mid?z → right!(z + b ×y) → X22)

X2 = (x21‖ X22)

显然

X2 sat (mid $\leq^1$a × leftl ∧ right $\leq^1$ mid + b × left2)

⇒ (right ≤ a× left1 + b × left2) □

X3 由左方输入数据流，向右方输出连续输入数对的加权和，权为 a 和 b。更精确地说，我们要求

right ≤ a× left + b × left'

只需在 X2 的解上增加新进程 X23，就可得到 X3 的解

X3 = (X2 ‖ X23)

其中

X23 sat (left1 $\leq^1$ left ∧ left2 $\leq^1$ left')

X23 可定义为

X23 = (left?x → leftl!x → (μX. left?x → left2!x → leftl!x → X))[221]

它向 left1 和 left2 通道拷贝来自 left 通道的数据，但第一个数据不向 left2 传送[222]。

这个的交叉网络 X3 的示意图可参见图 4.3 □

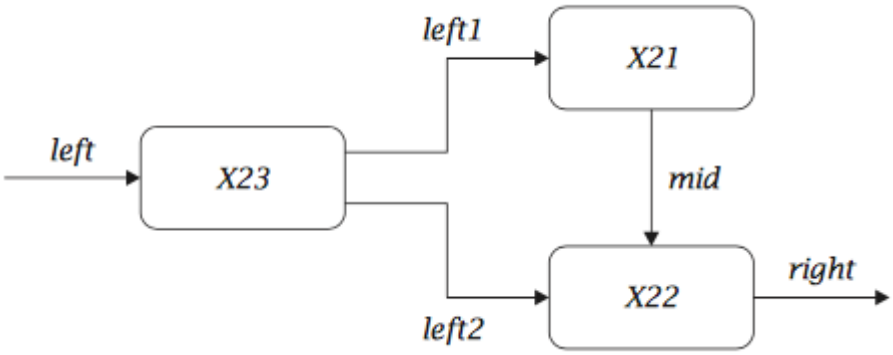


图 4.3

只在一个单一的通道上输入输出的、两个彼此通信的并发进程，是不会死锁的(参考 2.7 节 L2)。作为推论，非终止进程组成的无回路网络，也不会死锁，因为无回路图可分解为仅有一个箭头连接的子图。

但在网络 X3 中有无向回路，而回路网络分解为子网络时，必然会出现两个或两个以上通道相连的情况。对此类网络，不易保证其无死锁现象了。例如，如将 X23 的循环中的输出 left2!x → leftl!x →颠倒一下，马上就会出现死锁。

在证明无死锁时，往往可不必考虑传递的消息内容，因此可将通道 c 上的所有通信看做单一的事件，起名为 c。对非连接通道上的通信，也可不必理会。这样，可用这些事件改写 X3，得到

[221] X23 确保第一次首先和仅仅只把数据给 left1，从而通过 mid 可以点亮 X22。然后才开始正常的递归迭代的接收数据从 left 通道，然后同步分发给 left1 和 left2 两个通道。

[222] 因为第一次的时候，X21 的 mid 的数据还没有准备好。

$(\mu X \bullet leftl \rightarrow mid \rightarrow X)$

$\| (\mu Y \bullet left2 \rightarrow mid \rightarrow Y)$

$\| (leftl \rightarrow \mu Z \bullet left2 \rightarrow leftl \rightarrow Z))$

$= \mu X3 \bullet (leftl \rightarrow left2 \rightarrow mid \rightarrow X3)$

和 2.3 节 X1 一样，这里也使用代数方法，证明了 X3 不会死锁。

这些例子说明如何构造数据流网络，由一个或多个输入数据流，计算出一个或多个结果流。网络的形状和所计算的表达式中的运算对象和算子的结构紧密相关。当网络规模很大但保持一定规则时，我们使用带下标的通道名和多重并发组合的记号会带来很大便利

$\|_{i<n} P(i) = (P(0) \| P(l) \| ... \| P(n- 1))$

这类规则网络通常称作迭代阵列。若连接图中无有向回路，经常用术语脉动阵列来命名，因为数据通过这类系统很象血液通过心脏。

X4 通道$\{ left_j \mid j < n\}$上不断输入 n-维空间点的坐标。每个坐标集乘以长度为 n 的给定向量 V，并向右方输出所得的标量积；或更形式地表示为

$right \leq \sum_{j=0}^{n-1} V_j \times left_j$

在描述中还规定，每一微秒内，输入一个点的 n 个坐标，并输出一个标量积。每个处理器的速度近乎于，每一微妙内能做一次输入、一次乘法、一次加法和一次输出。为完成所描述的任务，显然需要至少 n 个处理器的并发操作。因此，需要设计一个不少于 n 个元素的迭代阵列，作为本问题的解决方法。

让我们用通常的归纳定义替代规约中的∑，有

$mid_0 = 0*$

$mid_{j+1} = V_j \times left_j + mid_j \qquad j<n$

$right = mid_n$

这样一来，我们将上述描述分割为 n+1 个简单方程，每个方程中至多包含一次乘法。剩下的事情就是对每个方程设计一个进程，对 j<n，令

$MULT_0 = (\mu X \bullet mid_0!0 \rightarrow X)$

$MULT_{j+1} = (\mu X \bullet left_j?x \rightarrow mid_j?y \rightarrow mid_{j+1}!(V_j \times x + y) \rightarrow X)$

$MULT_{n+1} = (\mu X \bullet mid_n?x \rightarrow right!x \rightarrow X)$

$NETWORK = \|_{j<n+2} MULT_j$

该阵列的连接图见图 4.4。

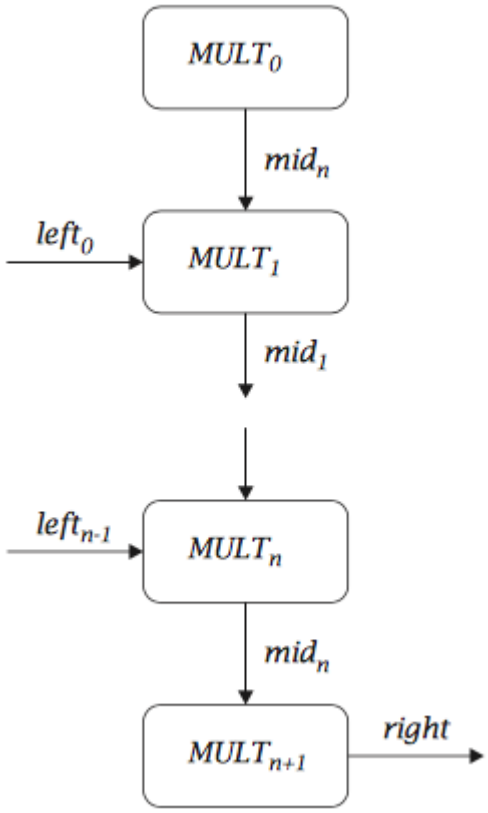


图 4.4

□

X5 本例类似于 X4，但要求由一个坐标集几乎同时得到 m 个不同的标量积。通道 $left_j$(j < n)用作输入一个无穷阵列的第 j 列；这个无穷阵列乘以给定的(n×m)矩阵 M，所得结果阵列的第 i 列经通道 $right_i$(i＜m)向外输出。用公式表示为

$$right_i = \sum_{j<n} M_{ij} \times left_j$$

结果的坐标值的输出速度，也要求如前例一般快速，故至少需要 m×n 个进程。

这个问题的解答可能能实际应用于图形显示设备，这种设备自动地变换，甚至旋转一个三维物体的二维表示。显示的形状由绝对空间中的一串点来确定的；通过迭代阵列，使用线性变换来计算阴极射线管 x 和 y 板极上的挠度；而输出的第三个坐标值可控制光线的强度。

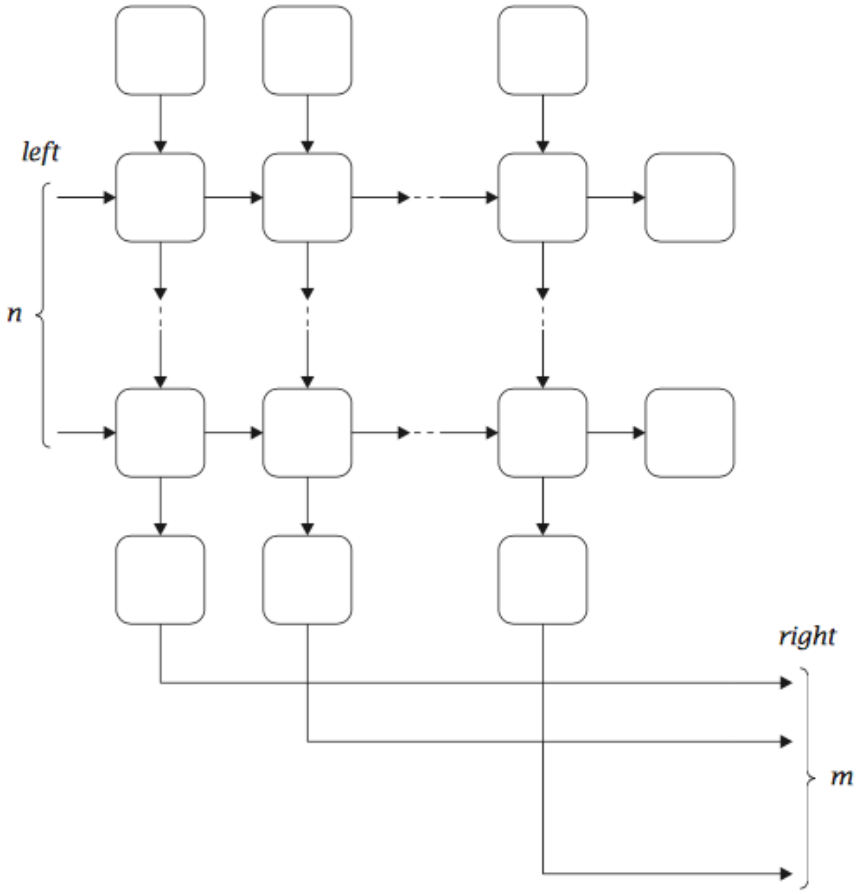


图 4.5

给出的解答基于图 4.5。除了最后一列，阵列的每列类同于 X4 的一个解；但它把由水平输入通道得到的数据，经水平输出通道，向其邻居交一份拷贝。最后一列的进程只会丢弃输入的值。为经济起见，可将该列的功能由其邻居代行。这个问题的具体方案细节留作练习给读者研究。 □

X6 通道 c 上相继输入的是自然数 C 的数位，由最小的有效数位开始，以 b 作为基数。定义输入数的值为

$$C = \sum_{i\geq 0} C[i] \times b^i$$

其中对一切 i 有 $c[i] < b$。

给定一个固定的乘数 M，由 d 通道相继输出乘积 M×C 的数位。要求结果数位输出的延迟极小。如果更精确地描述本问题，即要求输出 d 满足

$$d=\sum_{i\geq 0} M \times c[i] \times b^i$$

d 上的第 j 个元素必须是乘积的第 j 个数位，它的计算公式是

$$d[j] = (\sum_{i\geq 0} M \times c[i] \times b^i) \text{ div } b^j) \bmod b$$
$$= (M \times c[j] + z_j) \bmod b$$

这里 $z_j = (\sum M \times c[i] \times b^i) \div b^j$。div 表示整数除法。

z_j 是进位项，很容易可以证其明满足归纳定义

$Z_0 = 0$

$Z_{j+1} = ((M \times c[j] + Z_j) \text{ div } b)$

从而我们定义进程 MULT1(z)，它以进位 z 作为参量

MULT1(z) = c?x → d!(M × x + z) mod b → MULT1((M × x + z) div b)

z 的初始值是零，故所求答案为

MULT = MULT1(0) □

X7 本题同于 X6，只是 M 是一个多位数

$M = \sum_{i<n} M_i \times b^i$

单个处理器只能乘单个数位。而又要求产生输出的速度和一个数位的乘法速度相当。这样，就至少需要 n 个处理器。我们让每个 $NODE_i$ 照料乘数中的一个数位 M_i。

本题答案基于传统的多位数乘法的手写算法，不同处只是，表中的部分和立即加到下一行

...153091	C	the incoming number
253	M	the multiplier
...306182	$M_2 \times C$	computed by $NODE_2$
...765455	$M_1 \times C$	computed by $NODE_1$
...827275	$25 \times C$	
...459273	$M_0 \times C$	computed by $NODE_0$
...732023	$M \times C$	

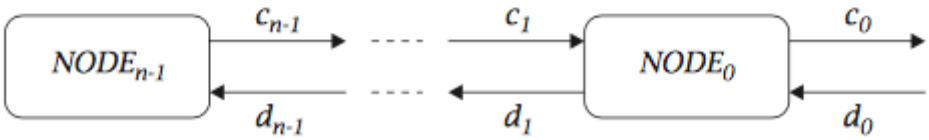


图 4.6

计算节点的连接见图 4.6.输入的原始数据由 C_0 通道进入，

然后经 c 通道向左传播。部分和是经 d 通道向右方传播，所要的乘积由 d_0 输出。每个节点在和其左邻通信前，就能给出一位乘积的一个数位。最左方的节点可定义为，如 X6 中答案般动作的一个进程，即

$$NODE_{n-1}(z) = c_{n-1}?x \rightarrow d_{n-1}!(M_{n-1}!\times x + z) \bmod b \rightarrow$$
$$NODE_{n-1}(M_{n-1}\times x + z) \operatorname{div} b)$$

其余节点与其类似，只是它们必须向左邻传递输入的数位，而且把左邻传来的结果加到自己的进位上。对 k＜n-1

$$NODE_k(z) = ck?x \rightarrow d_k!(M_k \times x + z) \bmod b \rightarrow$$
$$C_{k+1}!x \rightarrow d_{k+1}?y \rightarrow$$
$$NODE_k(y + (M_k \times x + z) \operatorname{div} b)$$

整个网络定义为

$$\|_{i<n} NODE_i(0)$$ □

X7 是一类精巧的网络算法中的一个简单例子，在这类算法中，通信通道的有向图中有一个不可缺少的回路。X7 中问题的陈述已大大简化了，假设乘数是事先知道的，而且永远固定不变。在实际应用中，更多情况下乘数亦是参量，和被乘的数据一道沿着同一通道输入进来；当要求改变乘数时，就重新输入。实现这个乘法时要很小心，但不要求更多的技巧。

一种简单的实现方法是，添加一个专用的符号，比如 reload，用这个符号指出，下面输入的数字或或数字串是用来改变参量的；如果参量的个数也是可变的，则还可能需要增加另一个符号 endreload，表示参量输入的结束。

X8 修改 X4，使参量 V_j 可以重新装填，由符号 reload 后紧跟的数字改变 V_j。$MULT_{j+1}$ 需要改成以乘数作为参量的进程

$$MULT_{j+1}(v) = left_j?x \rightarrow$$
$$\text{if } x = \text{reload then } (left_j?y \rightarrow MULT_{j+1}(y))$$
$$\text{else } (mid_j?y \rightarrow mid_{j+1} \times x + y) \rightarrow MULT_{j+1}(v))$$ □

4.4 管道(Pipe)

本节仅研究字母表中只含两个通道的进程，也就是一个输入通道 left，一个输出通道 right。这类进程称作管道，它们的示意图为图 4.7。

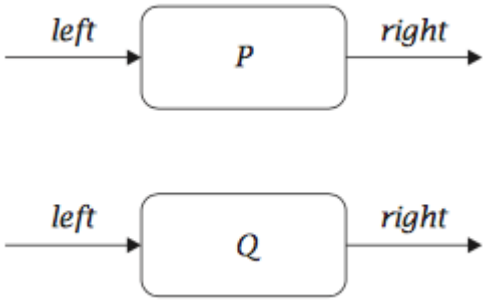


图 4.7

进程 P 和 Q 可以连接在一起，P 的右通道连向 Q 的左通道，在这条内部通道上，由 P 输出、Q 输入的消息序列被屏蔽起来，不被它们的环境所察觉。连接后的结果记作

P >> Q[223]

其示意图为图 4.8 中的链状结构。

图 4.8

这个连接图中不标明连接通道的名字，以此表示连接通道的屏蔽。它还表示了(P >> Q)左通道上的输入都来自 P，(P >> Q)右通道上的输出都来自 Q；另外，(P >> Q)本身是一个管道，它也可以和别的管道连成链，如

(P >> Q) >> R，(P >> Q) >> (R >> S), 等。

由 4.4.1 节的 L1，可知>>是满足结合律的，故以后在这种进程链中将不使用括号。

管道链接进程的正确性显然依赖于字母表的约束条件

$\alpha(P >> Q) = \alpha left(P) \cup \alpha right(Q)$

另外，连接的通道之间必须能够传送相同类型的消息是另一个保证正确性的约束条件

$\alpha right(P) = \alpha left(Q)$

例子

X1 将输入值乘以 4 再输出的管道(见 4.2 节 X2)

QUADRUPLE =DOUBLE >> DOUBLE

□

[223] 引入管道”>>“的符号和概念，是不考虑内部的通道，一种概念上的再次抽象。如果通过并发算子，其内部实现是:P = (left?x → mid!(x) → P)；Q = mid?y → right!(y) → Q)，然后 (P || Q)。

X2 输入 80 个字符的穿孔卡，并将它们组成每行 125 个字符的行式结构，成行输出(见 4.2 节 X3, X4)，即

UNPACK >> PACK

用通常的结构式程序设计的技术，设计这个进程是很困难的，因为弄不清主循环应该是，每输入一张穿孔卡重复一次，还是每输出一行循环一次。Michael Jackson 称之为结构冲突。上面给出的答案中，在两个进程中都分别包含有循环，和原来问题中的结构完全一致。 □

X3 同 X2，但把连续出现的星号兑换为“↑”(见 4.2 节 X5)

UNPACK >> SQUASH >> PACK

X3 和 X2 的描述只有很小的差别，但在传统的顺序程序设计中，这种差别会造成很多麻烦。为避开这个问题，这里使用了一个简单的妙法，即插入一个附加的进程。这类模块结构在设计操作系统时被提出和运用。

X4 同 X2，但当打印机不工作时，可连续输入穿孔卡；稍后，当读卡机不工作时，又可连续打印(见 4.2 节 X9)

UNPACK >> BUFFER >> PACK

缓存用来存储由进程 UNPACK 形成而尚且未被 PACK 进程消耗的字符。当 UNPACK 进程临时延误工作时，缓存可为 PACK 进程的输入服务。因之，缓存可用来调节产生和消耗的速率的断续变化。但它不可能解决产生和消耗的长时间的速率不匹配问题。如果读卡机的平均速度总慢于打印机，那末缓存几乎总是空的，无法调节。如果读卡机的平均速度总快于打印机，缓存将无限止地扩充。直到把所有的存储空间用光为止。

X5 无限缓存是无法实现的，为此我们通常限止所缓存的消息的个数。甚至采用 4.2 节中 X1 的 COPY 进程，这是一个单元的缓存。X4 答案的另一个版本是，在输入上可缓存一张穿孔卡，输出上可缓存一行字符

COPY >> UNPACK >> PACK >> COPY

注意，COPY 的前后两个实例的字母表是不同的，这可由它们所处的上下文看出。

X6 一个两单元的缓存，在输出第一个消息前至多可接受两个消息，可定义为

COPY >> COPY

它的行为类似于 CHAIN2(2.6 节 X4)和 VMS2(1.1.3 节 X6)。

4.4.1 法则(Laws)

链状进程的最有用的代数性质是结合性，即

L1 P >> (Q >> R) = (P >> Q) >> R

下面的其他法则用来说明，管道中的输入和输出是如何实现的；这些法则提供的符号运算，能用来简化进程的表述。例如，位于>>左方的进程，若一开始就要向右方输出消息 v，>>右方的进程，一开始就从左方输入消息，那末消息 v 就由前面的进程传到后面的进程；而且这个通信是被屏蔽的，故有

L2 (right!v → P) >> (left?y → Q(y)) = P >> Q(v)

若两个进程中的一个想与另一个通信，而另一个却准备与外界通信，那末，和外部的通信将首先发生，而内部通信将留待以后

L3 (right!x → P) >> (right!w → Q) =

right!w → ((right!v → P) >> Q)[224]

L4 (left?x → P(x)) >> (left?y → Q(y)) =

left?x → P(x) >> (left?y → Q(y)))[225]

若两个进程都要和外界通信，那末，哪一个都可能先发生

L5 (left?x → P(x)) >> (right!w → Q) =

(left?x → (P(x) >> (right!w → Q))

| right!w → ((left?x) → (P(x) >>Q))

法则 L5 中的算子>>换成>>R>>后，仍然成立；因为处于链的中央的管道是不能直接和外部环境通信的。

L6 (left?x → P(x)) >> R >> (right!w → Q) =

(left?x → (P(x) >> R >> (right!w → Q))

| right!w → ((left?x → P(x)) >> R >> Q))

法则 L6 的类似情况也可以推广到其它法则[226]

L7 若 R 是进程链，其中各进程均以向右输出作为开始，则

R >> (right!w → Q) = right!w → (R>>Q)

L8 若 R 是进程链，其中各进程均以从左方输入作为开始，则

(left?x → P(x)) >> R = left?x → (P(x) >> R)

例子

---

[224] Q 进程的 right!w 是一个对外的输出事件；P 进程的 right!v 则是一个内部的输出事件。

[225] P 进程的 left?x 是一个来自外部环境的输入事件；Q 进程的 left?x 则是一个内部的输入事件(来自 P)。

[226] L7 是 L3 的扩展；L8 是 L4 的扩展。

X1 让我们定义

R(y) = (right!y → COPY) >> COPY

则

R(y)

= (right!y → COPY) >> (left?x → right!x → COPY)　　COPY 的定义

= COPY >> (right!y → COPY)　　L2

□

X2 COPY >> COPY

= (left?x → right!x → COPY) >> COPY　　COPY 的定义

= left?x(right!x → COPY) >> COPY　　L4

=left?x → R(x)　　R(x)的定义

□

X3 由 X1 的最后一行可得

R(y) = (left?x → Right!x → COPY) >> (right!y → COPY)

= (left?x → (right!x → COPY) >> (right!y → COPY))

|right!y → (COPY>>COPY))　　L5

= (left?x → right!y → R(x)　　L3, X2

|right!y → left?x → R(x))

上述推导说明，一个两单元的缓存，在输入第一个消息后，既准备输出已输出的消息，也准备在输出第一个消息前再输入一个消息。上述证明的推理过程和 2.3.1 节的 X1 非常类似。 □

4.4.2 实现(Implementation)

在实现(P >> Q)时，要区分三种情形

1. 若在内部连接通道上可发送通信，则不考虑外部环境的影响，立刻实现内部通信。如果出现无穷的内部通信的序列，则该进程发散(参见 3.5.2 节)。

2. 不然，如果环境欲在左通道上通信，则由 P 处理。

3. 或者，环境愿在右通道上通信，则由 Q 处理。

至于输入和输出操作的实现，则参见 4.2.1 节。

```
chain(P, Q) =
    if P( "right) ≠"BLEEP∧Q( "left)≠"BLEEP then
        chain(cdr(P( "right)), Q( "left) (car(P( "right)) ) )        情形(1)
```

```
else
        λx •
                if x "right then
                        if Q( "right) = "BLEEP then
                                "BLEEP
                        else
                                cons(car(Q( "right)), chain(P, cdr(Q( "right)) ) 情形(2)
                else
                        if x = "left then
                                if P(x)= "BLEEP then
                                        "BLEEP
                                else
                                λy • chain (P("left)(y),Q)                    情形(3)
                        else  "BLEEP
```

4.4.3 活锁(Livelock)

链接(Chaining)算子用一条通道连接两个进程；因此不会导致死锁。如果进程 P 和 Q 都不终止，则进程(P>>Q)亦不会停止。可惜的是，链算子可能导致另外一种危险。即进程 P 和 Q 将所有的时间用于彼此通信，从而使(P>>Q)永不和外部世界通信。这类发散现象(参见 3.5.1 节和 3.8 节)可见于下列例子。最明显的是

P=(right!1 → P)

Q= (left?x → Q)

(P>>Q)显然是一个毫无用处的进程[227]；它甚至比 STOP 更糟，因为它是一个不做任何事情，而又消耗无穷无尽计算资源的，永不停止的循环。一个略微好一点的例子是

P = (right!1→ P | left?x → P1(x))

Q = (left?x → Q | right!1 → Q1)

在这个例子中，(P>>Q)的发散性来自存在无穷内部通信的可能性；尽管每次在 P 的左通道和 Q 的右通道上的外部通信的选项都是存在的，而且一经外部通信，(P>>Q)的后续行为就不会发散，但是这种无穷的内部通信的概率还是可能发生的。

[227] 无穷的内部动作。P 在右边的通道重复输出 1，不理会任何其他的环境事件，例如，左侧的输入通道。Q 则是仅仅反复的从左侧通道(来自 P 的右通道)读入数据，然后什么也不做，又递归回去读数据。

证明(P>>Q)无活锁的简单方法是证明 P 是左卫式(Left Guarded)的，也就是证明，P 在向右方输出无穷消息串的整个过程中，必须有时有从左方通道接收输入的通信事件。为保证这一点，我们需要证明，任何时刻向右方输出的序列的长度，都不超过左方输入(接收)的序列的某个函数值；更形式地，我们定义

P 是左卫的 ≡ ∃ f • P　　sat (#right ≤ f(left))[228]

由 P 的上下文经常不难证明其左卫性。

L1 P 的定义中的每个递归式，若都存在左方输入通道事件，则 P 是左卫的。

L2 若 P 是左卫的，则(P>>Q)无活锁。

上述推理完全适用于>>的第二个运算对象的右卫性[229]。

L3 若 Q 是右卫的，则(P>>Q)无活锁。

例子

X1 由 L1，下列进程(4.1 节 X1,X2，X5，X9)是左卫的

COPY, DOUBLE, SQUASH, BUFFER

X2 按照定义，下列进程也是左卫的，因为

UNPACK sat #right ≤ (^/left)

PACK sat #right ≤ #1eft

X3 BUFFER 不是右卫的，因为它在不向右方输出消息的情况下，也可以从左方输入任意多个消息[230]。

4.4.4 规约(Specifications)

管道的行为描述，通常可表示成左通道上输入的消息序列和右通道输出的消息序列间的关系，如 S(left, right)。当两个管道顺序相连时，左运算对象产生的 right 序列相等于右运算对象所消耗的 left 序列；然后这个共同的序列被屏蔽起来，对外在的环境不可观察。对这个共同序列，我们除了其存在性之外，别的一无所知。另外，我们还需避开活锁的危险。基于以上，我们有如下规则

L1 若 P sat S(left, right)

Q sat T(left, right)

且若 P 是左卫的，或 Q 是右卫的

[228] #right ≤ f(left)蕴含了存在进程从环境中接收输入的事件。而且是确定性的。

[229] 右卫(Right Guarded)的语义是一个进程会存在通过其右边的通道，向环境输出信息的事件。

[230] BUFFER 的定义可参阅 4.2 X9。其行为蕴含了可能无穷的接收左侧的数据，而不从右通道向环境输出。

则(P>>Q) sat ∃ s • S(left, s) ∧ T(s, right)

这个规则说明，(P>>Q) 所保持的 left 和 right 间的关系，就是 P 及 Q 所保持的关系的复合。由于算子>>不会导致死锁，故不必考虑拒绝集的推理。

例子

X1 DOUBLE sat right $\leq^1$ double* (left)

DOUBLE 即是左卫的，又是右卫的。

故 (DOUBLE >> DOUBLE)

sat ∃ s • (s $\leq^1$ double* (left) ∧ right $\leq^1$ double* (s))

≡ right $\leq^2$ double* (double* (left))

≡ right $\leq^2$ quadruple* (left)

X2 使用>>和递归式给出缓存的另一种定义

BUFF = μX • (left?x → X >> (right!x → COPY)))

我们希望证明

BUFF sat (right ≤ left)

假设

X sat #1eft ≥ n ∨ right ≤ left

我们知道

COPY sat right ≤left

故(right!x → COPY) sat ((right = left = < > ∨ (right ≥ x) ∧ right' ≤ left))

⇒ right ≤ <x>^left

由于右运算对象是右卫的，根据 L1 和假设

(X >> (right ! → COPY))

sat (∃ s • (#left ≥ n ∨ s ≤ left) ∧ right ≤ <x>^s)

⇒ #1eft ≥ n ∨ right ≤ (x) ^left)

因此

left?x → (...)

sat right =left =< > ∨

(left > <>) ∧ (#left' ≥n ∨ right ≤ <$left_0$>^left'))

⇒ #1eft ≥n + 1 ∨ right ≤ left

这样，由递归进程的证明规则(3.7.1 节 L8)，可得所求结论。

而 1.10.2 节中更简单的法则 L6 不能在这里使用，因为这个递归式显然不是卫式的。

4.4.5 缓存和协议(Buffers and protocols)

一个缓存是一种进程，它向右方输出从左方输入或者接收到的消息，输入和输出之间可能有某种延迟；当其非空时，还应时刻准备向其右方输出。可以形式化的描述为，一个缓存是一个进程 P，它永不终止，无活锁，而且满足描述

P sat (right ≤ left) ∧ (if right = left then left ∉ ref else right ∉ ref)

此处 c ∉ ref 意味着进程不能拒绝在通道 c 上进行通信(见 3.7 节和 3.4 节)。由上述描述可见，缓存是左卫式的。

例子

X1 下列进程是缓存

COPY, (COPY>>COPY), BUFF, BUFFER □

缓存显然可用于存储待加工的信息。更可以用作通信协议的行为的描述，通信协议在保持原有消息次序下传递消息。这类协议由两个进程组成，一个是发送方 T，另一个是接受方 R，它们顺序相连构成(T >> R)。如果这个协议是正确的，那(T >> R)显然必须是个缓存。

实际上，连接发送方和接收方的线路是非常长的，沿着这条线路传送消息时，可能出错或者丢失。线路本身的行为也可以用进程来模拟，称作 WIRE，但它的动作行为不同于缓存了。协议设计人员的任务就是要使这个系统，作为整体而言，仍如缓存一般动作，尽管线路可能产生出错或丢失不良行为；即保证

(T>>WIRE>>R))

是一个缓存。

一个协议通常由多层构成，如(T_1, R_1) , (T_2, R_2), ...，(T_n, R_n) ，每一层次以前面的一层作为通信介质，即

$Tn>>...>>(T_2>>(T_1>>WIRE>>R_1)>>R_2)>>...Rn$

当然，实际上实现一个协议时，将所有的发送进程集中起来成为单一个发送方，置于线路的一端；而将所有的接收进程集中起来，置于线路的另一端。这对应于下面定义式中括号的改变，写成

$(Tn>>...>>T_2>>T_1)>>$ WIRE $>>(R_1>>R_2)>>...Rn)$

>>满足结合律的性质，可以保证上述两种不同的分组方法不会改变系统的行为。

实际使用的协议比上述的协议复杂的多，因为单向的消息流是不适用的，是不可能在不可靠的线路上达到可靠的通信的；必须增加反向的通道，使接收方可以发回确认信号，通知发送方那些消息已成功接收，从而使未确认的消息可以重新发送。

下列法则对证明协议正确性是很有用的。它们是由 A.W.Roscoe 提出的。

L1 若 P 和 Q 是缓存，则(P>>Q) 和 (left?x → (P>>(right!x → Q)))也是缓存。

L2 若(T>>R) = (left?x → (T>>(right!x → R))) 则 (T>>R)是缓存。

L2 的一个推广如下

L3 若对某个函数 *f* 和对一切 z 有

(T(z)>>R(z))

= (left?x → (T(f(x, z)) >> (right!x → R(f(x,z)))))

则 T(z)>>R(z)对一切 z 都是缓存。

X2 由 L1 可知下述进程都是缓存

COPY>>COPY,BUFFER>>COPY,COPY>>BUFFER,BUFFE>>BUFFER

□

X3 如在 4.4.1 节 X1 和 X2 中已经指出的，

(COPY>>COPY) = (left?x → (COPY >> (right!y → COPY)))

由 L2，这也是一个缓存。

□

X4 (相位编码) 一个相位编码器是一个进程 T，T 输入数位流，并每输入一个 0 就输出数位对<1,0 >。译码器 R 则做相反的翻译。即

T = left?x → right!x → right!(1 - x) → T

R = left?x → left?y → if y = x then FAIL else (right!x → R)

其中的进程 FALL 未定义。

我们用 L2 证明(T>>R)是一个缓存

(T>>R)

=left?x → ((right!x → right!(1-x) → T) >>

(left?x → left?y →

if y = x then FAIL else (right!x → R)))

= left?x → (T>> if (1- x) = x then FAIL else (right!x → R))

=left?x → (T>>(right!x → R))

这样，由 L2 知(T>>R)是缓存。 □

X5 (数位填充) 发送端 T 忠实地将输入的数位由左向右转发，但每连续发送三个 1，就插入一个额外的数位 0。如输入时 01011110，则输出是 010111010。接收方 R 则删去这些多余的零。这样，(T>>R)必定是一个缓存。T 及 R 的构造，它们的正确性的证明，留作练习。 □

X6 (线路共享)要求将通道 left1 上的数据拷贝到 right1 上，将通道 left2 上的数据拷贝到 right2 上。为此，最容易的办法是用两个互不相干的协议，各自有专用的线路。可惜，现在只有一条可用的线路 mid，必须用于两个数据流，见图 4.9。

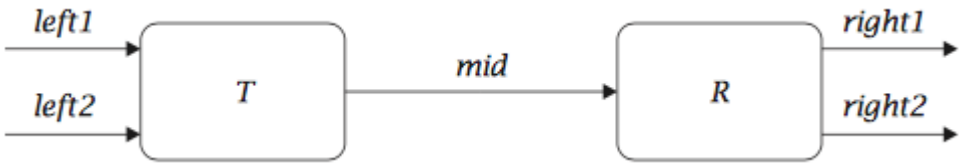


图 4.9

因此，在向 mid 传送输入的消息前，T 必须打标签；R 则删去标签，然后将加工后的消息由相应的右通道上输出[231]。即

T = (leftl?x → mid!tagl(x) → T
| left2?y → mid!tag2(y) → T)
R = mid?z →
if tag(z) =1 then
l!untag(z) → R)
else
(right2!untag(z) → R)

这个解是不能令人满意的。如果接连两个消息由 left1 输入，但相应的接收端尚未就绪，则全系统陷入等待，left2 和 right2 之间的传输只得推迟。在各通道上安置缓存，也不能彻底解决问题。正确的解决是引入一条反向通道，使 R 可向 T 发回信号，从而中止暂不需要的数据流。这种方法叫作流量控制。

4.5 从属(Subordination)

设 P 和 Q 为进程，且有

αP ⊆ αQ

[231] 读者可参阅网络系统里的 VLAN Tag 的例子。

并行组合(P ‖ Q)中，P 每做一个动作，必须得到 Q 的允准；而 Q 可独立执行(αQ-αP)中的动作，不必得到同伙 P 的批准，甚至也不必通知它。故 P 如同 Q 的奴隶或称从属进程，而 Q 则如主宰或称主进程。我们用非对称的记号

P // Q

表示从属进程和主进程的通信，这种内部通信对它们的外部环境而言是被屏蔽的，不可观察的。使用屏蔽算子，上述记号可定义为

P // Q = (P || Q) /αP

使用这个记号的前提 αP ⊆ αQ；因此

α(P // Q) = (αQ - αP)

为方便起见，通常给从属进程起个名字，比如叫 m，主进程就使用这个名字表示和其从属进程间的相互作用。2.6.2 节中说明的进程命名的技术不难扩充为通信进程命名，为此需引入复合通道名。复合通道名的形式为 m.c，其中 m 是进程名，c 是该进程的一个通道。在这种通道上的每个通信都表示为三元组

m.c.v

这里 αm.c(m:P) = αc(P)，v ∈ αc(P)。

在结构(m : P // Q)中，Q 通过复合通道 m.c 和 m.d 与 P 通信；而 P 用相应的简单通道 c 和 d 完成同样的通信。例如

(m : (c!v → P) // (m.c?x → Q (x))) = (m : P // Q(v))

由于这类内部通信全被屏蔽了，故外部环境不可能察觉进程名 m；这种名字可用作从属进程的局部名。从属进程可以嵌套，如

(n: (m : P // Q) // R)

在这种情况下，名字 n 只附加到出现在(aQ-aP)中的事件上；也就是在附加名字 n 前，先将涉及名字 m 的事件屏蔽掉。R 是无法和 P 直接通信的，甚至也无法知道 P 的存在，或者 P 的名字。

例子

X1 doub : DOUBLE // Q　　(关于 DOUBLE 可见 4.2 节 X2)

这里的从属进程好比是一个子程序，由主进程 Q 调用。在 Q 中，为得到值 2×e，需连续执行一次输出和一次输入：向 doub 的左通道输出参数 e，从其右通道输入结果。即

doub.left!e → (doub.right?x → ...)　　□

X2 子程序本身也可有子程序，如此这般可出现多重的子程序调用。令

QUADRUPLE

= (doub : DOUBLE // (u X • left?x → doub.left!x →
doub.right?y → doub.left!y →
doub.right?z → right!z X))

这个进程本身也可作为一个子程序

quad : QUADRUPLE // Q

QUADRUPLE 的这个版本类似于 4.4 节 X1，但不具有双倍缓冲功能。 □

X3 常见的程序变量也可通过从属进程模拟，设变量名为 m，则模拟作

m : VAR // Q

在主进程 Q 中，可使用输入和输出对 m 进行赋值，读出和更新，详见 2.6.2 节 X2。

m:= 3; P 通过(m.left!3 → P)实现

x:=m; P 通过 (m.right?x → P)实现

m:=m+3; P 通过(m.right?y → m.left!(y + 3) → P)实现 □

从属进程还可用来实现比简单变量更复杂的数据结构。

X4 (q : BUFFER // Q) (见 4.2 节 X9)

这个从属进程是一个名为 q 的无界队列。Q 用输出 q.left!V 向队列尾增添值 v,用输入 q.right?y 从队列头移走一个元素，并将其值赋予 y。当队列是空时，队列不响应后一操作，系统就可能死锁。 □

X5 名为 st 的堆栈

(st : STACK // Q) (见 4.2 节 X10)

在主进程 Q 中，st.left!v 可用于压入值 v，而 st. right?x 可弹出栈顶值。在取出栈顶值时，要处理可能出现的空栈情形，故可用选择结构

(st.right?x → Q1(x) | st.empty → Q2)

当堆栈非空时，选取第一种可能；空时，选取第二种可能，就避免了死锁。具有多个通道的从属进程可供多个并发进程调用，但要求各用不同的通道。 □

X6 进程 Q 向 R 传送一串值；并用从属的，名为 b 的缓存进程缓存传递的值；这样，当 R 的输入尚未就绪时，Q 的输出仍可进行。Q 使用通道 b.left 输出，而 R 使用通道 b.right 输入。构成

(b : BUFFER // (Q ‖R))

注意，即使 R 试图从空缓存输入消息，也不一定会造成系统死锁；只是 R 的这种要求不会立即满足，而要推迟到 Q 再输出一个值。(如果 Q 和 R 使用同一个通道和缓存通

信，那末这个通道必须在它们两者的字母表中出现；又由‖的定义，要求它们总是同时传递同样的数值。这就完全不符合我们的设计要求了。) □

从属算子亦可用于定义递归子程序。每层递归调用(最后一层除外)都会产生一个新的局部子程序。

X7 阶乘

FAC = μX • left?n →
(if n = 0 then
(right!1 → X)
else
(*f* : X // (*f*.left!(n-1) →
f.right?y → right!(n × y)→ X)))

子程序 FAC 使用通道 left 和 right 和其调用进程间传递参数和结果；使用 *f*.left 和 *f*.right 与它的从属进程 *f* 通信。从这些方面看，它类似于 X2 中的 QUADRUPLE 子程序。唯一的不同点是，FAC 的从属进程和它自身同构。 □

阶乘是一个很常见的的递归定义的例子，只是使用了一种陌生的、但比较麻烦的记号法重新表示了一下。用递归式和从属进程定义无界的数据结构会比较有新意一点。每个递归层次上，存储了数据结构的一个单一的成份，并且定义了一个新的局部数据结构来处理其他的成分。 □

X8 无界的有穷集

用进程来实现一个集合，它从左通道上输入元素。如果输入了一个已经输入过的元素，就输出 YES，否则输出 NO。这很象 2.6.2 节 X4 中的集合，但它可贮存任何种类的消息

SET = left?x → right!NO → (rest : SET // LOOP(x))

其中

LOOP(x) = μX • left?y →
(if y = x then
right!YES → X
else
(rest.left!y →
rest. right?z →
right!z → X))

开始时集合为空；输入第一个元素 x 后，立即输出 NO。然后建立一个从属进程 rest，它将贮存集合中除 x 外的其它元素。LOOP 是用于集合的其它元素的输入处理。若新输入的元素等于 x，立即由右通道上发回 YES。否则将新元素传给 rest，由它贮存起来。由 rest 发回的回答(YES 或者 NO)也照传不误；LOOP 不断重复上述过程。

X9 二叉树

用二叉树可得到集合的另一种效率更高的表示方法，二叉树依赖于元素间的某种全序≤。每个节点上保存最早登入的元素，并且建立两个从属树，一个用于贮存小于这个最先元素的元素，另一个则贮存大于它的元素。树的外部描述同于 X8.令

TREE = left?x → right!NO → (smaller : TREE //(bigger : TREE //LOOP))

LOOP 的设计则留作练习。

4.5.1 法则(Laws)

主进程及其从属进程间通信的法则如下。第一个法则说明主进程及从属进程间每个方向上的通信都被屏蔽

L1A (m : (c?x → P(x))) // (m.c!v → Q) = (m : P(v)) // Q

L1B (m : (d!v → P)) // (m.d?x → Q (x)) = (m : P) // Q(v)

若 b 为不带有名字 m 的通道，则主进程可在不影响从属进程情况下，在 b 上通信

L2 (m : P // (b!e → Q)) = (b!e → (m : P // Q))

只有主进程才能对从属进程做出相应的选择

L3 (m : (c?x → P1(x) | d?y → P2(y))) // (m.c!v → Q) = (m : PI(v) // Q)

两个同名的从属进程中，必有一个是不可能调用的

L4 m:p //(m:Q//R) = (m:Q//R)

从属进程的书写次序通常是无所谓的。

L5 若 m 和 n 是不同的名字

m:P//(n:Q//R) = n:Q//(m:P//R)

使用递归式定义从属进程时，会使人很吃惊，无法肯定所定义的进程能否正常工作。为减少这种疑虑，需要说明所定义的进程是如何动作的。下面的例子中，说明一个特定的进程行为迹是如何产生的。更重要的是，它还说明稍有改动的迹为何就不能产生了。

例子

X1 SET 的一个典型迹是

s = <left.1, right.NO, left.2, right.NO>

使用 L1A, L1B 和 L2，经一系列步骤，可算出 SET/s 的值

SET / (left.l) = right!NO → (rest : SET // LOOP(1))

SET / (left. 1, right.NO) = (rest : SET // LOOP(1))

SET / (left. 1, right.NO, left. 2)

= (rest : SET //rest.left!2 → rest.right?z → Right!z → LOOP(1)))

= (rest : (right!NO→ (rest : SET // LOOP(2))) //

(rest.right?z → right!z → LOOP(1)))

= rest : (rest : SET // LOOP(2)) // (right!NO → LOOP(1))

因此

SET / s = rest : (rest : SET // LOOP(2)) // LOOP(1)

从上述推导显然可见

<left. 1, right.NO, left.2, right. YES>不是 SET 的迹。

读者可以验证

SET / s^ <left.2, right.YES) = SET / s

及

SET /s^ <left.5, right.NO> =

rest : (rest : (rest : SET // LOOP(5)) // LOOP(2)) // LOOP(1) □

4.5.2 连接图(Connection diagrams)

用方框表示具有从属进程的系统，则从属进程画在方框的内部，图 4.10 是 4.5 节 X1 的示意图。

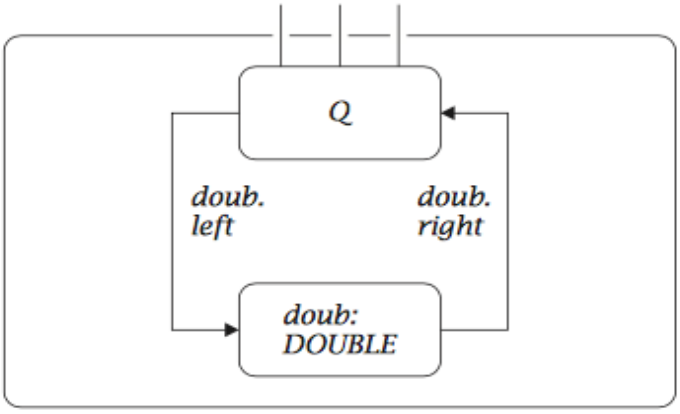


图 4.10

在嵌套从属进程的示意图中，方框间也多重镶嵌，如 4.5 节 X2 的图 4.11 示意图。

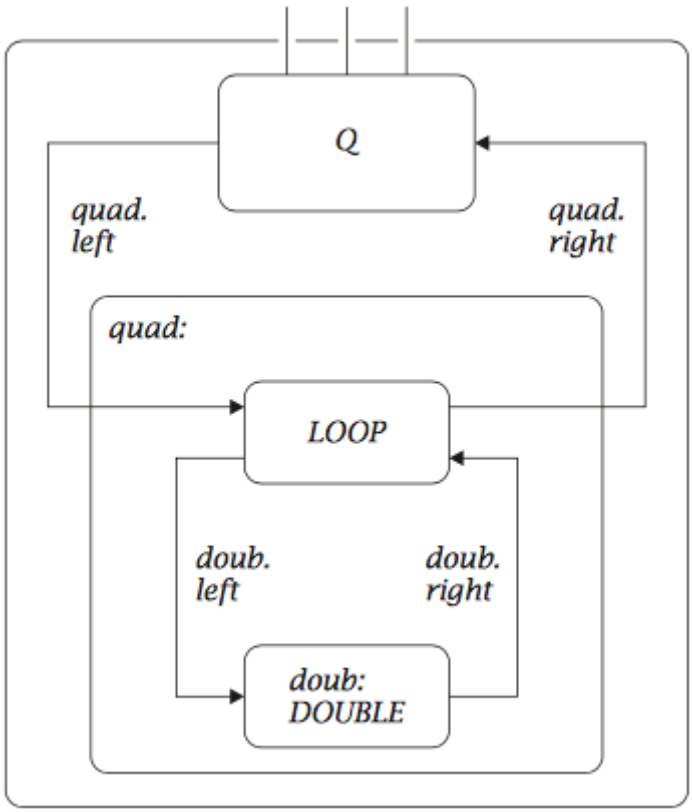


图 4.11

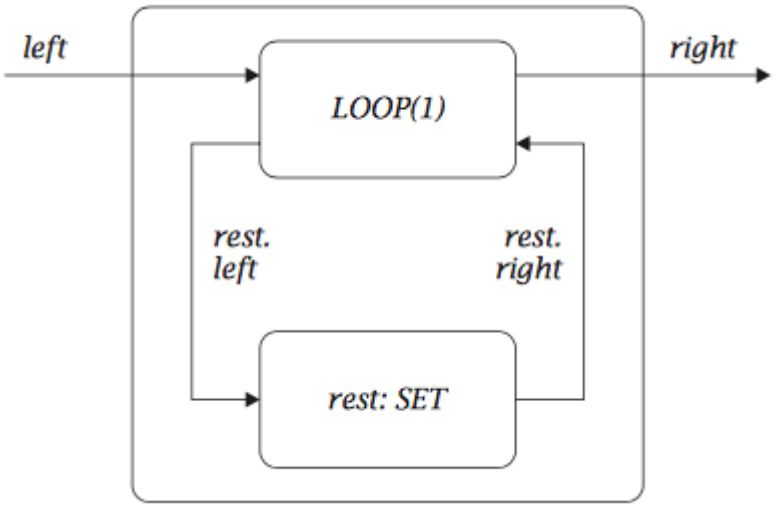
一个递归进程是自嵌套的一幅图，就象一幅关于画家工作室的画，在工作室的画架上挂着这幅完成了的画，而这幅画里面又有一个画架挂着一幅完成了的画，.....。实际上是无法画完这幅画的。好在对于进程而言，不需要一幅完成的图，只要说明在它的活动过程中，它是如何按人们要求自动地展开的。这样，4.5.1 节 X1 中的集合的早期的活动史可分阶段地表示在图 4.12 中。

如对方框的嵌套图不感兴趣，则可用线性结构表示集合，见图 4.13.类似地，TREE(4.5 节 X9)可画作图 4.14。

连接图告诉人们如何由硬部件构成相应的网络，其中的方框代表集成线路，箭头代表集成电路间的连线。在实现递归时，当然必须先将递归展开到一个有限的层次，然后才能启动有关网络；如果在网络操作过程中，越出了给定的层次界限，那么网络就不能继续工作。传统顺序程序设计中实现递归的有效方法是堆栈式存储分配，通过硬件网络的动态分配和重构来实现递归就比较困难。递归存在的价值在于，它有

助于算法的发明和设计，或者至少给那些能够理解和使用递归的人呆了智力上的快乐。

SET / (left.l, right.NO) =



SET / s =

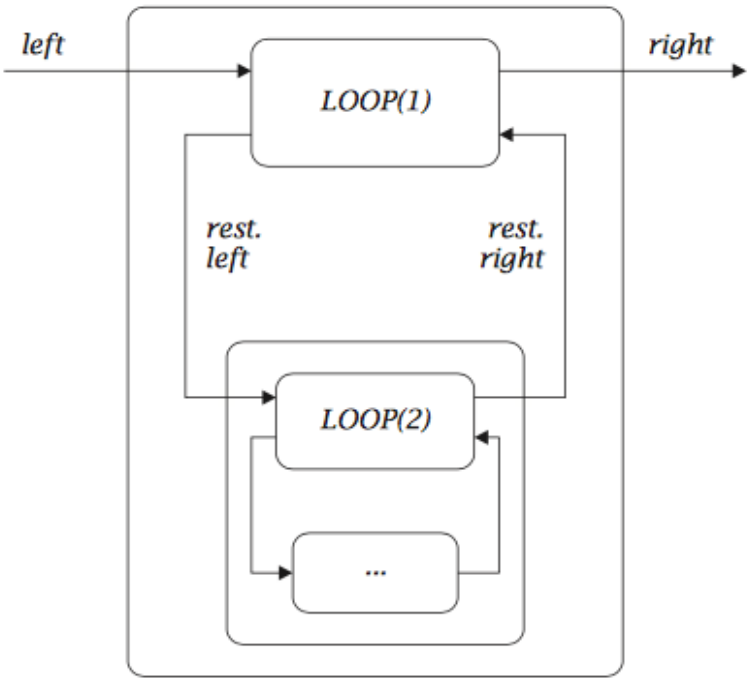


图 4.12

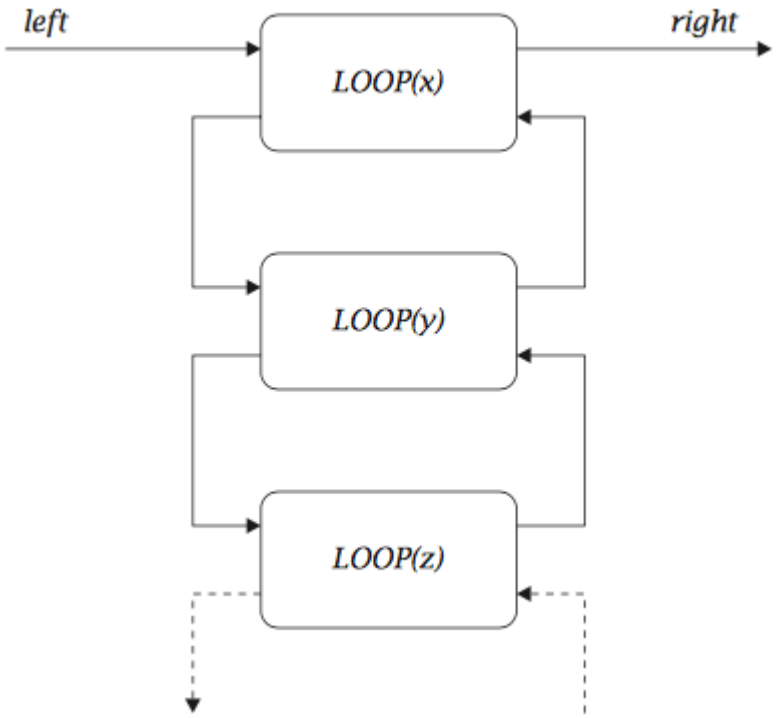


图 4.13

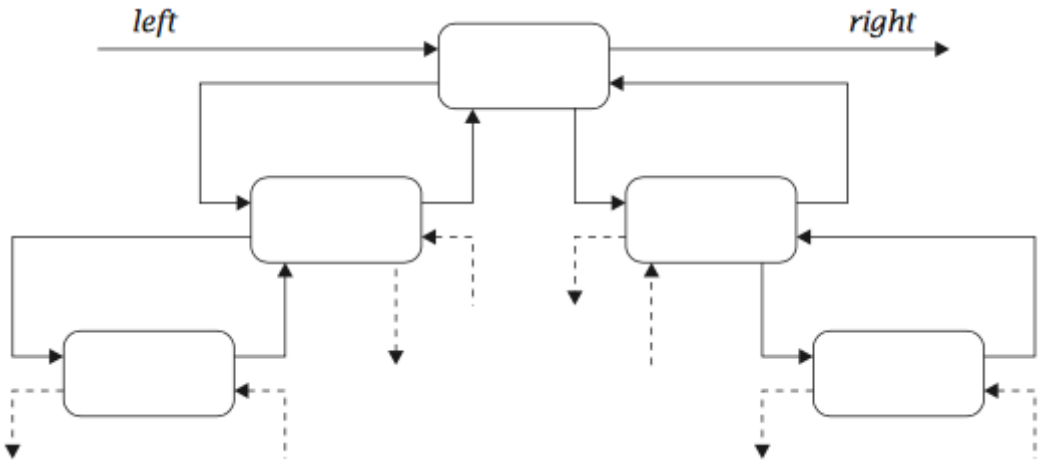


图 4.14

第五章　顺序进程(SEQUENTIAL PROCESSES)

5.1 引言(Introduction)

STOP 是一个从不执行任何动作的进程[232]。这个进程没有多大用处，多半不是设计者有意选用的，而是死锁现象或设计错误造成的。然而，当一个进程已经完成设计交付的任务时，它就有理由停下来不去做任何更多其他的事情[233]。这时称这个进程成功地终止了。为了区分 STOP 和成功终止，我们将成功终止看作是一种特殊的事件，用符号 √记之(称之为"Success")。本书中，我们定义顺序进程(Sequential Processes)是字母表中包含有 √的一种进程；而且√是其执行的最后一个事件。因此，我们规定在选择结构中不允许出现 √，即若 $\surd \in B$，则

(x: b → P(x))是无效的。

$SKIP_A$是一个都不做，只表示成功终止的进程，有 $\alpha SKIP_A = A \cup \{\surd\}$

和往常一样，我们通常略写字母表下标。

例子

X1 只向一位顾客出售巧克力或太妃糖，然后就终止的售后机为

VMONE = (coin → (choc → SKIP | toffee → SKIP))　　□

在设计一个进程解决一个复杂任务时，往往将该任务分成为两个子任务，其中之一先执行，只有当它成功终止后，才启动另一子任务。设 P 和 Q 是具有相同字母表的顺序进程，它们的顺序组合

P ; Q

亦是一个进程。这个进程先按 P 动作；当 P 成功终止后，(P ; Q)接着执行 Q 的动作。如果 P 永远不终止，则(P ; Q)也不终止[234]。

X2 顺序地，但只为两位顾客服务的售货机

VMTWO = VMONE ; VMONE　　□

一个顺序进程，按要求不断重复类似行为，称作循环进程；可以认为是递归的一种特例。我们给出其定义如下

$*P = \mu X \bullet (P ; X)$

---

[232] 不响应任何来自环境的事件触发，既不接收，也不输出。

[233] 进程行为里不存在递归。

[234] 如果按照定义，"顺序组合"算子的定义是 P 和 Q 本身需要是一个顺序进程，需要能够成功终止。

= P ; P ; P; …

α(*P) = αP - {√}

很明显，这种循环永不成功终止；这也就是从其字母表中删除√的原因。

X3 可为任意多个顾客服务的售货机

VMCT = *VMONE

□

这和 1.1.3 节 X3 中的 VMCT 完全一样[235]。

若进程 P 执行一系列的动作后终止，这串动作所对应的符号序列就称作是进程 P 的一个句子(sentence)。P 的所有句子的集合称为 P 所接受的语言(language)。故用来描绘顺序进程的符号约定可用来定义简单语言的文法，这种语言也许可用作人机间的通信。

X4 一个语言称作 Pidgingol，其句子是由名词子句及其后随的谓语组成的。一个谓语则是跟有名词子句的动词。名词子句及其它成分的定义形式地给出如下

αPIDGINGOL = {a, the, cat, dog, bites, scratches}

PIDGINGOL = NOUNCLAUSE ; PREDICATE

PREDICATE = VERB ; NOUNCLAUSE

VERB = (bites → SKIP | scratches → SKIP)

NOUNCLAUSE = ARTICLE ; NOUN

ARTICLE = (a → SKIP| the → SKIP)

NOUN = (cat → SKIP | dog → SKIP)

Pidgingol 的句子如

the cat scratches

a dog a dog bites the cat

□

为了描述一个具有无穷多个句子的语言，必须使用某种迭代或递归的方式。

X5 名词子句中可包含任意多个形容词 furry 或者 prize

NOUNCLAUSE = ARTICLE ; μX • (furry → X | prize→ X

| cat → SKIP | dog → SKIP)

这类名词子句如

the furry furry prize dog

a dog

[235] 在 1.1.3 X3 的定义里，是通过一个递归定义来完成无线的，并停机的售货机。VMCT = μ X • coin → (choc → X | toffee → X) 。这里的 VMCT 是通过 LOOP 的概念，在每一次会正常终止的顺序进程的概念下，定义了无数个串联的顺序进程组，从而实现无限不停止的行为。

□

X6 一个进程接受任意多个 a，然后一个 b，然后和 a 个数相同多的 c，最后成功终止。令

$$A^nBC^n = \mu X \bullet (b \rightarrow SKIP \mid a \rightarrow (X ; (c \rightarrow SKIP)))$$

若先接受 b，则进程终止；此时，即未接受 a，也未接受 c，故它们的个数相同都是零。若取第二个分支，其所接受的句子则以 a 打头，以 c 结尾，中间是递归调用进程 X 而接受的句子。如果我们假定递归调用接受的 a 和 c 的个数相同，那末 A^nBC^n 的非递归调用也接受等量的 a 和 c，其接受的句子比之递归调用所接受的句子，开头多个 a，结尾多个 c。这个例子表明如何使用顺序组合的递归式，来定义具有无穷多个状态的机器。 □

X7 定义一个进程，先如 A^nBC^n 般动作，然后接受一个 d，接着再接受等量个数的 e。令

$$A^nBC^nDE^n = ((A^nBC^n); d \rightarrow SKIP) \parallel C^nDE^n$$

其中

$C^nDE^n = f(A^nBC^n)$ f 将 a 映射为 c, b 为 d, 及 c 为 e。

这个例子中，‖的左边的进程负责保证 a 和 c 的个数相同，且中间由 b 分开。在接受适量个数 c 以后，才接受一个 d；但它不接受 e(e 不在它的字母表中)。‖的右边的进程负责保证 e 的个数和 c 一样。它不理会 a 和 b，因为它们不在它的字母表中。这时进程在完成分派的任务后，一起终止。 □

接受进程定义语言的能力和正则表达式的能力相同。使用递归，可增加上下文无关文法的部分能力，但不是其全部。进程所能定义的语言必须能由左至右实现语法分解，分解时不回溯也不超前。其原因在于，在使用选择算子时要求各个分支的第一个事件都不相同。这样，在 X5 中就不可能定义名词子句，使字 prize 即可为名词又可为形容词，例如 the prize dog, the furry prize。使用算子□ (见 3.3 节)也无济于事，因为这会引入非确定性，导致可以任选一个子句来分析剩下的输入。一旦选错了，就会酿成死锁，无法读遍输入的全部内容。解决这一问题，需要一种新型的选择算子，它能象

or3(3.2.2 节)一样，提供安琪儿非确定性[236]。在环境做出选择前，这种选择算子的两个分支一直并发运行；其定义留作读者练习。

上述定义语言的方法，在没有安琪儿非确定性选择算子情况下，其能力不如上下文无关交法，因为它要求自左向右的可分解性，而且不允许回溯。但是，引入‖算子后，可定义非上下文无关的语言，例如 X7。

X8 可以接受 down 和 up 任意穿插的序列的进程，并要求 down 的个数一旦超出 up 的个数时，进程就终止。即

$$POS = (down \rightarrow SKIP \mid up \rightarrow (POS ; POS))$$

若第一个符号是 down，POS 的任务就完成了。若第一个符号是 up，则必须再多接受两个 down。为此，先设法多接受一个 down，然后再多接受一个。故需要顺序地两次递归调用 POS。 □

X9 下列进程 C_0 的行为类似于 CT_0(1.1.4 节 X2)

$$Co = (around \rightarrow Co \mid up\ C_l)$$

$$C_{n+l} = POS ; C_n$$

$$= POS ; ...; POS ;\ POS ;\ POS ; C_0 \qquad n \geq 0$$ □

我们现在可以给出 2.6.2 节 X3 中所提出问题的一个解决方案，这个问题在 4.5 节 X3 中也提到过。解决这个问题用到对从属进程的操作，必须清楚地说明用户进程中这个操作之后的其余动作。使用 SKIP 和顺序组合后，就可方便地得到所要求的结果。

X10 进程 USER 操作两个计数变量 m 和 l(参见 2.6.2 节 X3)，即

$$l : CT_0 \,\|\, m : CT_3 \,\|\, USER$$

下列子进程(属于 USER)将 l 逐步减少，所减掉的值加到 m 上(利用递归调用 ADD)。然后 m 再增加一次，l 也增加一次，这样可补偿最初减少的值，使 l 恢复为初始值。

□

5.2 法则(Laws)

顺序组合的法则类同于连接(见 1.6.1 节)，SKIP 是基本单元。

L1 $SKIP ; P = P ; SKIP = P$

L2 $(P ; Q) ; R = P ; (Q ; R)$

L3 $(x{:}B \rightarrow P(x)) ; Q = (x{:}B \rightarrow P(x) ; Q))$

选择算子法则的推论是

[236]安琪儿非确定性(Angelic nondeterminism)是指非确定性进程的一种实现方式，一个进程不自己做非确定性的选择，留着给运行时环境决定。

L4 $(a \rightarrow P) ; Q = a \rightarrow (P ; Q)$

L5 $STOP ; Q = STOP$

将顺序进程并行组合起来，只有当两个进程都终止时，整个组合式才成功终止，即

L6 $SKIP_A \| SKIP_B = SKIP_{A \cup B}$

成功终止的进程不再参予并发进程提供的任何事件

L7 $((x : B \rightarrow P(x)) \| SKIP_A) = (x : (B - A) \rightarrow (P(x) \| SKIP_A))$

顺序进程与非顺序进程的并发组合，什么情况下才会成功终止呢？如果顺序进程的字母表完全包含了另外合作进程的字母表，那么组合式的终止性由顺序进程所决定；其原因是，当顺序进程结束后，另一个进程也什么都做不成了。故有

L8 $STOP_A \| SKIP_B = SKIP_B$ 若$\surd \notin A$ 且 $A \subseteq B$

这条法则的前提条件是很合理的，当并发运行的进程对中只有一个进程的字母表包含$\surd$时，该前提条件必须遵守。因为，只有这样才能避免一个进程在执行$\surd$后还继续活动。

法则 L1 和 L3 可用来证明 5.1 节 X9 中的断言，即 C_0 和 CT_0(1.1.4 节 X2)的行为相同。为此，需要说明 C 满足定义 CT 的卫式递归方程组。首先，定义 CT_0 的方程和定义 C_0 的方程相同。

$C_0 = (around \rightarrow C_0 \mid up \rightarrow C_1$ C_0 定义

对 $n>0$，需证明

$C_n = (up \rightarrow C_{n+1} \mid down \rightarrow C_{n-1}$

证明

LHS

$= POS ; C_{n-1}$ Cn 定义

$= (down \rightarrow SKIP \mid up \rightarrow POS ; POS) ; C_{n-1}$ POS 定义

$= (down \rightarrow (SKIP ; C_{n-1}) \mid up \rightarrow (POS ; POS) ; C_{n-1}$ L3

$= (down \rightarrow C_{n-1} \mid up \rightarrow POS ; (POS ; C_{n-1}))$ L1, L2

$= (down \rightarrow C_{n-1} \mid up \rightarrow POS ; Cn)$ Cn 定义

$= RHS$ C_n 定义

由于 C_n 遵从的卫式递归方程组和 CT_n 的相同，故它们的行为也相同。

为了阐明上述法则的应用，也免得怀疑为循环推理，在上面这个证明中，我们把细节都写的很全。这个证明最让人疑惑的是证明中没有使用关于 n 的归纳法。实际

上，由于 CT_n 的定义中包含了进程 CT_{n+1}，使用对 n 的归纳法都不会成功的。好在关于解的唯一性的法则，很简单明了。

5.3 数学处理(Mathematic treatment)

在形式描述顺序组合的数学定义时，应该保证上一节中列出的法则的真实性。特别要留意的是

P ; SKIP = P

和往常一样，确定性进程的数学处理比较简单，因此由此先开始。

5.3.1 确定性进程(Deterministic processes)

我们用进程的迹来定义确定性进程的运算的结果。SKIP 进程的第一个也是仅有的一个动作是成功终止，故它只有两个迹

L0 traces(SKIP) = {< >, <√>}[237]

为定义进程顺序组合，我们先定义进程迹的顺序组合。若 s 和 t 是迹，s 不包含√，则

(s ; t) = s

(s^<√>) ; t = s^t

详见 1.9.7 节。(P ; Q)的迹是由 P 的迹所组成，且当 P 的迹以√结尾时，将√替换为 Q 的迹，即

L1 traces(P ; Q) = {s; t | s ∈ traces(P) ∧ t ∈ traces(Q) }

与其等价的定义是

LIA traces(P ; Q) = {s | s ∈ traces(P) ∧ ¬<√> in s} ∪

{s^t | s^ <√ > ∈ traces(P) ∧ t ∈ traces(Q) }

这个定义易于理解，但难以使用。

当进程执行 √后，进程应该终止，这是符号 √的全部含义。故我们需要下列法则

L2 P / s = SKIP 若 s^<√ > ∈ traces(P)

上述这个法则对于证明

P ; SKIP = P

是必不可少的。可惜，该法则只对部分进程有效。例如，令

$P = (SKIP_{\{\}} \parallel c \rightarrow STOP_{\{c\}})$

[237] 读者可参阅 1.8 关于 STOP 进程的迹的定义。traces(STOP) = {<>}。

则 traces(P) = {< >, <√>, <c>, <c, √>, <√ c>}[238] ,尽管

<√> ∈ traces(P)，但 P/< > ≠ SKIP。因此，需要对并行组合的字母表加以约束。只有当

$$\alpha P \subseteq \alpha Q \vee \alpha Q \subseteq \alpha P \vee \surd \in (\alpha P \cap \alpha Q \cup \alpha P \cap \alpha Q)$$

成立时，(P ‖ Q)才是有效的组合。

同样的原因，在变换字母表时，不能变换√，故只有

$$f(\surd) = \surd$$

时，f(P)才有效。

若 m 是进程名，我们还需要规定

$$m.\surd = \surd$$

最后，我们不允许在选择结构中使用√，如

$$\surd \rightarrow P \mid \surd \rightarrow Q$$

由于这种限制，当√ ∈ A 时，RUN_A 也就不合法了。

5.3.2 非确定性进程(Non-deterministic processes)

非确定性进程的顺序组合带来一系列问题。首先，非确定进程不满足上节的法则 L2，如 SKIP ⊓ (c → SKIP)。

解决这个问题的一种办法是减弱 5.3.1 节中的 L2，变为

L2A $s^{\frown}\langle\surd\rangle \in traces(P) \Rightarrow (P/s) \subseteq SKIP$

这条法则表明，当 P 能终止的时候，P 可以不再向环境提供可能事件，进入终止状态。

为使 L2A 成立，必须遵守前面各节中的限制，而且还要规定

SKIP 不能无卫地出现在□的运算对象中

√ 不能出现在|||的运算对象的字母表中

(如果改变一下□和|||的定义，上述限制可有所放松。)

除了本章中已给出的法则外，非确定性进程的顺序组合还满足下述法则。首先，不论如何规定进程终止后的行为，发散进程仍发散，即

L1 CHAOS ; P = CHAOS

顺序组合对非确定性选择满足分配律。

L2A (P ⊓ Q) ; R = (P ; R) ⊓ (Q ; R)

[238]并发组合的两个进程 SKIP 和(c → STOP)的字母表是完全不同的。因此 P 的迹是两个进程迹的混合。

L2B R; (P ⊓ Q) = (R ; P) ⊓ (R ; Q)

在非确定性进程的数学模型范畴下(3.9 节)定义(P ; Q)，需要处理它的失败集和发散性。我们先刻画它的拒绝集(见 3.4 节)。若 P 拒绝 X，又不能成功终止，则 X∪<√> 也是 P 的拒绝集(见 3.4.1 节 L11)。在这种情况下，X 也是(P ; Q)的拒绝集。但 P 若可能终止，那末在(P ; Q)中可自主地出现顺序过渡；而且这种过渡动作是被屏蔽的，这样，Q 的拒绝集也成了(P ; Q)的拒绝集。P 的终止性也可能是非确定的，这一情形也应考虑。故有定义

D1 refusals(P ; Q)= {X| (X ∪ {√}) ∈ refusals(P) }

∪ {X | <√> ∈ traces(P) ∧ X ∈ refusals(Q) }

(P ; Q)的迹的定义对于确定的或者非确定性的进程都是相同的。(P ; Q)的发散定义为：当 P 发散，或者 P 成功终止而 Q 发散时，(P ; Q)发散。

D2 divergences(P ; Q) ={s | s ∈ divergences(P) ∧ ¬<√> in s} ∪ {s^t | s^ ∈ traces(P)

∧ ¬<√> in s ∧ t ∈ divergences(Q) }

(P ; Q)的任何失败集，要末是 P 在终止前的一个失败，或者是在 P 终止后 Q 的一个失败，即

D3 failures(P ; Q) ={ (s, X) | (s, X ∪<√>) ∈ failures(P) } ∪

{ (s^t, X) | s^<√> ∈ traces(P) ∧(t, X) ∈ failures(Q) } ∪

{ (s, X) | s ∈ divergences(P ; Q) }

5.3.3 实现(Implementation)

SKIP 可由下述进程来实现，它只接受符号"SUCCESS。至于它以后的动作则无关紧要。令

SKIP = λx • if x = "SUCCESS then STOP else "BLEEP

对于顺序进程的组合，若第一个运算对象终止，则顺序组合的行为同第二个运算对象；否则，由第一个运算对象参予第一个事件，余下的部分再和第二个运算对象组合。

sequence(P, Q) = if P("SUCCESS) ≠ "BLEEP then Q

else λx • if P(x) = "BLEEP then "BLEEP

else sequence(P(x), Q)

5.4 中断(Interrupts)

在本节中，我们定义另一种顺序组合(P △ Q)，这种顺序组合不要求 P 的终止性。当 Q 的第一个事件一出现，P 就被中断；而且 P 一旦中断，就不再继续。故(P△Q)的迹就是 P 至中断点的迹，再跟上 Q 的迹，有

$\alpha(P \triangle Q) = \alpha P \cup \alpha Q$

$traces(P \triangle Q) = \{ s^\frown t \mid s \in traces(P) \wedge t \in traces(Q) \}$

为避免一些问题，我们规定 αP 不能包含√。

下面的法则说明，Q 的启动是由环境决定的；它通过选择一个 Q 的初始事件，而不是 P 的事件，来启动 Q，即

L1 $(x:B \rightarrow P(x)) \triangle Q = Q \,\square\, (x:B \rightarrow P(x) \triangle Q)$

(P △ Q)被 R 中断同于 P 被(Q △ R)中断，有

L2 $(P \triangle Q) \triangle R = P \triangle (Q \triangle R)$

由于 STOP 不提供任何事件，故不能被环境所驱动。类似地，若 STOP 被某个进程中断，则只会出现中断进程的动作。故 STOP 是△的单位元，即

L3 $P \triangle STOP = P = STOP \triangle P$

另外，中断算子同时执行其两个运算对象，故它对非确定性选择算子满足分配律，有

L4A $P \triangle (Q \sqcap R) = (P \triangle Q) \sqcap (P \triangle R)$

L4B $(Q \sqcap R) \triangle P = (Q \triangle P) \sqcap (R \triangle P)$

最后，中断一个发散进程不能阻止发散；发散进程中断一个其它进程也是不安全的，会发散下去。规则可表示为

L5 $CHAOS \triangle P = CHAOS = P \triangle CHAOS$

本节的余下部分中，我们规定中断进程的初始事件不属于被中断进程的字母表。由于中断现象是环境可观察到的，而且环境可以影响控制，这样就不会引入非确定性，还可简化有关推理。为着重讨论确定性的保持问题，我们扩充原有的选择算子。假设设 c ∈ B，我们有

$(x : B \rightarrow P(x) \mid c \rightarrow Q) \equiv (x : (B \cup \{c\}) \rightarrow (\text{if } x = c \text{ then } Q \text{ else } P(x)))$

类似地也可定义运算对象更多的情形。

5.4.1 灾难(Catastrophe)

令符号↯表示一种灾难性中断事件，自然 P 不应发生这种事件；形式地定义为，

$$\lightning \notin \alpha P$$

在发生灾难前，行为如 P，然后如 Q 的进程，定义为

$$P \mathbin{\hat{\lightning}} Q = P \triangle (\lightning \rightarrow Q)$$

其中的 Q 是用于从灾难中恢复的进程[239]。注意，算子$\hat{\lightning}$和事件$\lightning$是不同性质的两个符号。

下面的第一个法则是将$\hat{\lightning}$算子的非形式化描述公式化

L1 $(P \mathbin{\hat{\lightning}} Q) / (s^{\wedge}\langle\lightning\rangle) = Q \qquad s \in traces(P)$

在确定性模型中，这个法则唯一地定义了算子的含义。但在非确定性范畴里，要唯一地定义算子的语义，还需要陈述它的严格性和满足分配律。

第二个法则更清晰地阐述这个进程的第一步动作及其后的各步动作。它表现为$\hat{\lightning}$如何对$\rightarrow$自后分配的

L2 $(x:B \rightarrow P((x)) \mathbin{\hat{\lightning}} Q = (x:B \rightarrow (P(x) \mathbin{\hat{\lightning}} Q \mid \lightning \rightarrow Q)$

这条法则对于确定性进程而言是多余的，第一条法则已唯一地定义了这个算子对确定性进程的效用。

5.4.2 重启(Restart)

对灾难的一种可能的反应是重新启动原来的进程。令 P 是进程，$\lightning \notin \alpha P$。我们定义$\hat{P}$为一个新进程，在$\lightning$事件出现前按照 P 的行为动作，在每次$\lightning$中断出现后再由 P 从头执行。这种进程称为可再启动进程。这个行为可通过简单的递归定义为

$$\alpha\hat{P} = \alpha P \cup \{\lightning\}$$

$$\hat{P} = \mu X \bullet (P \mathbin{\hat{\lightning}} X)$$

$$= P \mathbin{\hat{\lightning}} (P \mathbin{\hat{\lightning}} (P \mathbin{\hat{\lightning}} ...))$$

递归式中 X 都是通过事件$\lightning$为前缀卫式保护的，因此是卫式递归方程。虽然 P 不是循环进程，$\hat{P}$是一个循环进程(见 1.8.3 节)。

使用重新启动算子不仅是为了从灾难中恢复。例如，假设一个游戏进程，其对手是人，通过选用键盘上的按键实现双方的交互作用(参见 1.4 节中的函数 interact)。有时，游戏者不满意于游戏的进展情况，可能希望重新启动。为此，在键盘上应提供一个新的专用键($\lightning$)；在游戏过程中，一按下此键，就可重新启动一盘新的游戏。这样就

[239] Q 可以理解为中断处理程序。

可以，不考虑重新启动功能，先独立地设计游戏 P，然后使用上面定义的算子，将其转换成可重新启动的游戏$\hat{P}$。这个想法是由 Alex Teruel 提出的。

P 的非形式定义可用下列法则表示

L1 $\hat{P}$/ s^<↯> = $\hat{P}$ $\quad$ s ∈ traces(P)

上述这条法则不能唯一地定义 P，因为 RUN 进程也一样满足这条法则。但$\hat{P}$是满足 L1 的最小的确定性进程。

5.4.3 交替(Alternation)

假设 P 和 Q 是两个游戏进程，其行为类似 5.4.2 节所描述；有一个人类棋手想同时玩这两个游戏，交替地进行，如同国际象棋大师同时巡回地与几个弱手对弈。为此，我们再提供一个新按键⊗，用它可使得在 P 和 Q 之间交替进行。这种现象有点象中断，区别是这个被中断的游戏的当前状态可以保存起来，一旦另一个游戏之后被中断后，就可恢复运行前一个。同时玩这两个游戏的进程记作(P⊗Q)，并可由下列法则清楚地描述为

L1 ⊗ ∈ (α(P ⊗ Q)) - αP - αQ)[240]

L2 (P ⊗ Q) / s = (P / s) ⊗ Q 若 s ∈ traces(P)

L3 (P ⊗ Q) /<⊗> = (Q ⊗ P)

我们要的是满足 L2 和 L3 的最小进程。这个算子的更构造式的刻画可由上述法则推导出来；它表现为 是如何对→反向分配的

L4 (x:B → P((x)) ⊗ Q = (x:B → (P(x) ⊗ Q | ⊗ → (Q ⊗ P))

交替算子的用途不光可用于游戏中。一个“友善”操作系统中，应可交替使用各种系统功能，故需要类似的机制。例如，当你在使用编辑程序时，你可能要转到 help 程序，寻求系统帮助，但你绝不会希望回去时，失去你在编辑程序中已编辑的地方，反之亦然。

5.4.4 检查点(Checkpoints)

进程 P 描述一个可以存储持久化数据的数据库系统。当遇到电击中断时(↯)，最坏的可能就从初始状态开始重新执行 P，但这样会丢失了辛苦积累的系统数据。比较

[240] 这个定义似乎有点问题。 ⊗ 如果是事件，就不应该是算子。需要引入另外一个符号。类似重启(Restart)里对事件和算子的区别。

好的办法是返回到系统之前的某个近期状态，而且这个状态又可令人满意的[241]。这种状态叫作一个检查点。因此，我们再提供一个新按键©遇到满意的状态时，就可按下此键。在出现↯时，可以恢复最新一次的检查点；如果没有检查点，则恢复到初始状态。我们假设©和↯不属于 P 的字母表，定义进程 Ch(P)，其行为如 P，但对这两个事件以适当的方式分别响应。

Ch(P)的非形式定义可非常简洁地由下列法则形式化

L1 Ch(P) / (s^<↯>) = Ch(P)　　s ∈ traces(P)

L2 Ch(P) / (s^<©> = Ch(P/s)　　s ∈ traces(P)

Ch(P)可由两元算子 Ch2(P, Q)定义，其中 P 是当前的进程，Q 是等待恢复的最近的检查点。若灾难出现在第一个检查点之前，则系统就重新启动，详见下面的法则

L3 Ch(P) = Ch2(P, P)

L4 若 If P = (x:B → P(x))

则 Ch2(P, Q) = (x : B → Ch2(P(x), Q)

| ↯ → Ch2(Q, Q)

| © → Ch2(P, P))

法则 L4 对实际的实现方法很有启发，实际实现时，将检查点状态贮存在便宜又耐久的介质上，如磁盘或磁带上。当出现©事件时，就将当前状态拷贝成为新的检查点；如果出现↯，检查点的数据被拷贝回去作为新的当前状态。从经济上考虑，系统实现者总是设法使当前状态和检查点状态二者间共享尽可能多的数据。这种优化问题很大程度上依赖于具体的机器和应用；令人高兴的是其所用到的数学基础其实非常简单。

检查点算子不仅可用于大规模数据库系统。在玩一个复杂的游戏时，人们可能想先探究可用的策略，但不愿在尝试过程中真的执行。他就按一下©键，贮存当前的状态，然后再做试探，若试探失败，就用↯键恢复状态。

检查点的想法是由 Ian Hayes 考察和研究的。

[241] “满意的状态”通常意味是一个一致性满足了的状态。

5.4.5 多重检查点(Multiple checkpoints)

在使用可检查的系统 Ch(P)时，可能发生检查点出错的情况。这时，就会删去最近的检查点，而回溯到前面一个检查点。这就要求系统保留两个或更多近期检查点。原则上说，没有理由不让设计一个系统 Mch(P)，使其保持自运行以来的所有检查点。每当出现↯，就返回到最近的©的前面的那个状态，而不是其后面的状态。我们仍要求

αMch(P)- αP = {©, ↯}

出现在©之前的↯，使进程从头开始

L1 Mch(P)/s^<↯> = Mch(P)　　s ∈ traces(P)

出现在一个©之后的↯，会清除了最临近的©以来(包括这个©本身)所发生的一切事情对进程状态的影响。

L2 Mch(P) / s^<©>^t<↯> = Mch(P) / s　　(s ⌈ (αP)^t ∈ traces(P)

Mch(P)的行为可以使用二元算子 Mch2(P, Q)更好的表达，其中 P 是当前进程，Q 是存有等待恢复的检查点的堆栈。堆栈的初值是 P 的无穷多个拷贝

L3 Mch(P)= μX • Mch2(P,X)

= Mch2(P, Mch(P))

= Mch2(P, Mch2(P, Mch2(P, ...)))

在出现 ©时，当前状态就压入堆栈；出现↯，恢复整个堆栈。有

L4 若 P = (x：B → P(x))

则 Mch2(P,Q)=(x:B → Mch2(P(X),Q | © → Mch2(P, Mch2(P,Q)) | → 空 Q)

L4 中的递归很精巧，但当检查点的个数很大时，实现多重检查点的代价很大。

5.4.6 实现(Implementation)

中断的各种实现都是通过基于算子对→的分配律法则。考虑 5.4.3 节 L4 中的交替算子，可以有如下

```
alternation(P, Q) = λx • if x= ⊗ then
                        alternation( Q, P)
                    else if P(x) = "BLEEP then
                        "BLEEP
                        else
                            alternation(P(x), Q)
```

Mch(5.4.5 节 L3，L4)的实现更令人惊奇

Mch(P) = Mch2(P, Mch(P))

其中

Mch2(P, Q) = λx • if x = ↯ then

Q[242]

else if x = © then

Mch2(P, Mch2(P, Q))

else if P(x) = "BLEEP then

"BLEEP

else

Mch2(P(x), Q)

执行上面这个函数时，所用的存储量随检查点的个数成比例地增长；存储空间会很快被用完。尽管可用内存回收程序(Gabarage Collection)，从而每次出现↯后，使一些存储空间获得释放并重新使用，但这不能解决根本问题。在递归的实际实现中，一般都设置约束条件，规定递归深度的界限。在 Mch 算子方面，设计者也应对保存的检查点的数目作出限制，将早期的检查点扔掉。但用递归来表示这种设计就不很漂亮了。

5.5 赋值(Assignment)

本节中将引入传统的顺序程序设计中最重要的算子，即赋值，条件和循环。为简化某些法则的形式表示，我们将定义一些不常见的符号记法。

传统计算机程序设计最本质的特征是赋值。若 x 是一个程序变量，e 是表达式，P 是进程，则

(x:=e; P)

也是一个进程，其行为如 P，但 x 的初值为表达式 e 的初值。其余变量的初值不变。赋值本身的含义可定义为

(x:= e) = (x:=e; SKIP)

单个赋值可容易地推广为多重赋值。令 x 代表一列不同的变量。

$x = x_o, x_l, \ldots x_{n-1}$

e 代表表达式的表

$e = e_o, e_l, \ldots e_{n-1}$

假设这两个表(list)的长度相同，则

[242] 在 CSP 1985 年印刷版本的书里，此处公式为 Q。但 Hoare 的 2015 年的电子版为 Mch2(P, Mch2(P,Q))。应该是 Q 才正确。

x := e

将 e_i 的的初值赋予每个相应的 $x_i(0\leq i\leq n-1)$。注意，所有的 e_i 的求值先于任意赋值，故若 y 在表达式 g 中出现，则

y := f; z := g

不同于

y, z := f, g

令 b 是取值为布尔值(true 或 false)的表达式，P 和 Q 是进程，则

P ⊲ b ⊳ Q　　(P if b else Q)

是一个进程，当 b 初值为真时，其行为如 P；b 初值为假时，行为如 Q。这个记法是新颖的，比传统的记法 if b then P else Q 简洁。

类似的原因，传统的循环 while b do Q 将写成

b * Q

可用递归定义为

D1 b * Q = μX • ((Q; X) ⊲ b ⊳ SKIP)

例子

X1 一个进程，其行为如 CTn(1.1.54 节 X2)。

X1 = μX • (around → X | up → (n := 1 ; X))

⊲ n=0 ⊳

(up → (n := n + 1 ; X) | down → (n := n -1 ; X))

变量 n 记录了计数器的当前值。 □

X2 行为如 CT_0 的进程

n:=0; X1

计算器的初值设置成零。 □

X3 行为如 POS(5.1 节 X8)的进程

n：=1; (n > 0) * (up → n := n + 1 | down → n := n-1

用循环代替了原来的递归式。 □

X4 一个进程，以正数 y 除自然数 x，将商赋予 q，余数赋予 r。

QUOT = (q := x ÷ y ; r := x - q × y)[243] □

X5 和 X4 的功能相同，但用减法计算商，即

LONGQUOT = (q := 0; r :=x; ((r≥ y) * (q := q+1; r := r - y))) □

[243] Hoare 2015 CSP 电子书里是 q := x + y。应该是笔误。正确的应该是 q := x ÷ y 。

在 4.5 节 X3 中，我们已说明，变量的行为可用从属进程来模拟，从属进程和使用它的进程传递变量值[244]。本章中，我们有意地避免使用这种技术，因为它不具有我们所要的数学性质。譬如，我们希望

(m :=1; m :=1) = (m :=1)

遗憾的是

(m.left!1 → m.left!1 → SKIP) ≠ (m.left!l → SKIP)

5.5.1 法则(Laws)

在赋值法则中，x, y 代表不同的变量的列表；e, f(x), f(e)代表表达式的列表，它们可能包含 x 或 y 中的变量；f(e)包含 e_i，f(x)包含 x_i。为简单起见，我们假定下列法则中的所有表达式都是全函数，即对变量的任何可能值，都能给出结果。

L1 (x := x) = SKIP

L2 (x := e; x := f(x)) = (x := f(e))

L3 若 x, y 是不同变量的一个列表，则(x := e) = (x, y := e, y)

L4 若 x, y, z 是长度和 e, f, g 分别相同的列表，则(x, y, z := e, f, g) = (x, z, y :=e, g, f)

使用这些法则，可以将一系列的赋值转化为对所有有关变量的表的单个赋值。将≮b≯看作二元的中缀算子，它具有几个熟知的代数性质

L5-6 ≮b≯是幂等、结合，且对≮c≯满足分配律

L7 P ≮true ≯ Q = P

L8 P ≮false ≯ Q = Q

L9 P ≮ ¬b ≯ Q = Q ≮ b ≯ P

L10 P ≮ b ≯ (Q ≮ b ≯ R) = P ≮ b ≯ R

L11 P ≮ (a ≮ b ≯ c) ≯ Q= (P ≮ a ≯ Q) ≮b ≯ (P ≮ c ≯ Q)

L12 x := e ; (P ≮ b(x) ≯ Q) = (x := e ; P) ≮ b(e) ≯ (x := e ; Q)

L13 (P ≮ b ≯Q) ; R = (P ; R) ≮ b ≯ (Q ; R)

为有效地处理并发进程中的赋值，必须保证在一个并发进程中赋值的变量不会在另一个进程中使用。在顺序进程的字母表中，我们引入两类新的符号

var(P)可在 P 中赋值的变量集

acc(P)可在 P 中的表达式中可能被访问的变量集

[244] 这是个很重要的观点，即程序原因的变量可以通过 CSP 理论里的基本概念进程来实现。

可被赋值的变量也一定是可能被访问

$$var(P) \subseteq acc(P) \subseteq \alpha P$$

类似地，我们将出现在 e 中的变量记作 acc(e)。设 P 和 Q 由‖连接，我们规定

$$var(P) \cap acc(Q) = var(Q) \cap acc(P) = \{\}$$

在这个条件下，先于并行分叉赋值，或后于并发运行在一个并发成分中赋值，都是一样的，即

L14 $((x := e\ ;\ P) \| Q) = (x := e;\ (P \| Q))$

其中 $x \subseteq var(P) - acc(Q)$ and $acc(e) \cap var(Q) = \{\}$

我们可以得到一个显然的推论如下：

$$(x{:=}e;\ P)\ ||\ (y{:=}f\ ;Q) = (x, y := e,f;\ (P\ ||\ Q))$$

其中 $x \subseteq var(P) - acc(Q) - acc(f)$，且 $y \subseteq var(Q) - acc(P) - acc(e)$。

上述法则说明，通过字母表的限制足以保证并发进程的不同分进程的赋值不会彼此干扰。在实现时，可将赋值集中执行或者任意穿插执行，而不会影响进程的外部可观察的行为。

最后一个法则是，并发组合对条件算子满足分配律

L15 $P \| (Q \lhd b \rhd R) = (P \| Q) \lhd b \rhd (P \| R)$ $acc(b) \cap var(P) = \{\}$

这条法则再一次表明，b 的值在并行分叉前或后都相同。

我们现在讨论表达式对某些变量值无定义的问题。若 e 是表达式表，令 e 为一个布尔表达式，其取真值当且仅当 e 中所有运算对象的值都在相应算子的定义域中。例如，在自然数的算术中，

$$\text{ᗰ}(x \div y) = (y > 0)$$

$$\text{ᗰ}(y + 1, z + y) = true$$

$$\text{ᗰ}(e + f) = \text{ᗰ}e \wedge \text{ᗰ}f$$

$$\text{ᗰ}(r - y) = y \leq r$$

我们有理由认为 ᗰe 总是有定义的，即 ᗰ(ᗰe)=true

我们有意根本不提无定义表达式求值的结果，认为这样做可能引发任何事情。我们在法则中用 CHAOS 反映这个观点。

L16′ $(x := e) = (x := e \lhd \text{ᗰ}e \rhd CHAOS)$

L17′ $P \lessdot b \gtrdot Q = ((P \lessdot b \gtrdot Q) \lessdot \text{ⅅ} b \gtrdot CHAOS)$

法则 L2，L4 也需作小的修改

L2′ $(x := e; x := f(x) = (x := f(e) \lessdot \text{ⅅ} e \gtrdot CHAOS)$

L4′ $(P \lessdot b \gtrdot P) = (P \lessdot \text{ⅅ} b \gtrdot CHAOS)$

5.5.2 规约(Specification)

顺序进程的规约，不仅要刻画可能出现的事件的迹，而且要刻画程序变量的初始值、终止值以及迹之间的关系。我们用程序变量名字 x 表示程序变量 x 的初始值。而用 x 加上上标√，即 $x^{\surd}$，表示 $x^{\surd}$的值。因此，只$\underline{tr}_0$=√时，才能对 $x^{\surd}$进行描述。

例子

X1 什么事都不做，只把 x 加 1，y 不变，然后成功终止的进程其描述为

$tr = <> \vee (tr = <\surd> \wedge x^{\surd} = x+1 \wedge y^{\surd} = y =$ □

X2 只做一个事件的进程，这个事件的符号就是变量 x 的初值，做完这个事件就终止，x 和 y 的终止值同于初始值，其描述为

$tr = <> \vee tr = <x> \vee (tr = <x, \surd> \wedge x^{\surd} = x \wedge y^{\surd} = y)$ □

X3 将第一个事件赋予 x，成为其终止值，y 不变，描述为

$\#tr \le 2 \wedge (\#tr = 2 \Rightarrow (tr = <x^{\surd}, \surd> \wedge y^{\surd} = y))$ □

X4 用正数 y 除非负数 x，将商赋予 q，余数赋予 r，即

$DIV = (y > 0 \Rightarrow$

$tr = <> \vee (tr = <\surd> \wedge q^{\surd} = (x \div y) \wedge$

$r^{\surd} = x - (q^{\surd} \times y) \wedge y^{\surd} = y \wedge x^{\surd} = x))$

如去掉前置条件，这个描述是不可能满足的。 □

X5 如下是一个更复杂的规约，以后会用到。

$DIVLOOP =$

$(tr = <> \vee (tr = <\surd> \wedge r = (q^{\surd} - q) \times y + r^{\surd} \wedge$

$r^{\surd} < y \wedge x^{\surd} = x \wedge y^{\surd} = y))$

$T(n) = r < n \times y$

这里使用的(包括出现在以后描述中的)变量都表示自然数，故在第二个操作数大于第一个操作数时，减法是无定义的。

现在我们来描述用来证明进程满足其规约的法则。令 S(x, tr, $x^{\surd}$)是一个规约。为证明 SKIP 满足这个描述，显然要求当迹为空时，这个描述为真；而且当迹为<√>，$x^{\surd}$的值等于其初始值时，描述亦成立。这两个条件就是 SKIP 满足该描述的充分条件，如下列法则所示

L1 若 S(x, <>, $x^{\surd}$)

且 S(x, <√>, x)

则 SKIP sat S(x, tr, $x^{\surd}$)

X6 SKIP 所满足的最强规约是

$SKIP_A$ sat (tr =< > ∨(tr = (√) ∧ $x^{\surd}$=x))

其中 x 是 A 中所有变量的列表， $x^{\surd}$是这些带勾变量的表。X6 可由 L1 直接推论，反之亦然。 □

X7 SKIP sat (r < y ⇒ (T(n + 1) ⇒DIVLOOP))

证明

(1) 将规约中的 tr 代之<>，得到

r < y ∧ T(n+1) ⇒ <> = <> ∨...

这是一个重言式。

(2) 将描述中的 tr 代之<√>，终止值代之初始值，得到

r < y ∧ T(n+1) ⇒ (<√> = <> ∨ (<√> = <√> ∧ x = x ∧

y = y ∧ r = ((q - q) × y + r ∧ r < y)))

这也是一个显而易见的定理。其结论将用于 X10。 □

表达式 e 有定义是一个有效赋值 x:=e 的前置条件。在这个前置条件下，若 P 满足描述 S(x)，修改 S(x)使 x 的初值是 e，则(x :=e; P)满足修改后的描述。

L2 若 P sat S(x)

则(x := e; P) sat (𝔇e ⇒ S(e))

将 L2 中 P 代之以 SKIP 并使用 X6 及 5.2 节 L1，就可导出简单赋值语句的法则。

L2A x_0 := e sat (𝔇e ∧ tr ≠ <> ⇒ tr = <√> ∧ $x_0^{\surd}$ = e ∧ $x_1^{\surd}$ = x_1 ∧ ...)

作为 L2 的推论，对任意 P，关于(x :=1/0; P)，能证明的最强事实是

(x := 1/0; P) sat true

如果你想达到任何一个真正的目标，你就不能从非法赋值开始。

X8 SKIP sat (tr ≠<> ⇒(tr = <√> ∧ $q^{√}$ = q ∧ $r^{√}$ = r ∧ $y^{√}$ = y ∧ $x^{√}$ = x)

因此

(r := x - q×y; SKIP) sat (x≥q×y ∧ tr ≠ <> ⇒

tr = <√> ∧ $q^{√}$ = q ∧

$r^{√}$ = x - q × y ∧ $y^{√}$= y ∧ $x^{√}$=x

故

(q := x÷y; r :=x - q × y) sat (y＞0 ∧ x≥(x ÷ y) × y ∧ tr ≠ <> ⇒

tr = <√> ∧ $q^{√}$ = (x ÷ y) ∧ $r^{√}$ = (x - (x ÷ y) × y) ∧

$y^{√}$ = y ∧ $x^{√}$ = x)

上一行中的描述等价于定义在 X4 中的 DIV。 □

X9 假设

X sat (Γ(n) ⇒ DIVLOOP)

则

(r：= r - y ; X) sat (y≤r ⇒

(r-y<n×y ⇒

(tr =<> ∨ tr =< √>∧(r-y)=...)))

故

(q:=q+1;r:=r -y ; X) sat (y≤r⇒(r < (n + 1) ×y ⇒ DIVLOOP'))

其中

DIVLOOP' =

(tr = <> ∨ tr = <√> ∧ (r - y) = $q^{√}$-(q + 1)) × y + $r^{√}$ ∧

$r^{√}$< y ∧ $x^{√}$= x ∧ $y^{√}$= y)))

由自然数的初等代数

y ≤ r ⇒ (DIVLOOP' ≡ DIVLOOP)

故

(q := q+1； r := r - y; X) sat (y ≤ r ⇒ (Γ(n + 1) ⇒ DIVLOOP))

上面这个结论将在 X10 中用到。 □

对于一般的顺序组合，需要更为复杂的法则，不仅要将各组成部分的迹顺序地组合起来，而且第二个成分的初始状态其实就是等价于第一个组成进程的终止状态。

但是，在这中间状态中，变量的值是观察不到的；我们只能保证这样的值是存在的，即

L3 若 P sat S(x，tr，$x^{\surd}$)

且 Q sat T(x，tr，$x^{\surd}$)

并有 P 不发散

则

(P; Q) sat (∃ y, s, t • tr = (s ; t) ∧ S(x, s, y) ∧ T(y, t, $x^{\surd}$))

在上述法则中，x 是 P 和 Q 字母表中全体变量组成的列表，$x^{\surd}$是它们带下标后组成的表，y 是与它们个数相同的新变量的表。

条件算子的规约，当条件为真时，等同于第一个组成进程的规约；条件为假时，则同于第二个组成进程的行为描述，即

L4 若 P sat S，且 Q sat T 则

(P ◁ b ▷ Q) sat ((b ∧ S) ∨ (¬b ∧ T))

这条法则的另一可能便于使用的形式是

L4A 若 P sat(b ⇒ S)，且 Q sat(¬b ⇒ S)则

(P ◁ b ▷ Q) sat S

X10 令 COND = (q := q + 1; r := r -y ; X) ◁ r ≥ y ▷ SKIP

且

X sat (T(n) ⇒ DIVLOOP)

则

COND sat (T(n + 1) ⇒ DIVLOOP)

上面结论所需的两个充分条件，已在 X7 和 X9 中证明，故由 L4A 得到所求结论的证明。 □

为证明循环算子的特性，需用到 5.5 节的递归定义 D1，和无卫式递归的法则(见 3.7.1 节 L8)。若 R 是所要的循环进程的行为描述，则我们必须找到一个规约 S(n),使得 S(0)恒真，且

(∀ n • S(n)) ⇒ R

构造 S(n)的一个通用方法是找到一个谓词 T(n, x)，用来描述初始状态为 x 时，循环进程在 n 次重复前终止的条件。然后定义

S(n) = (T(n, x) ⇒ R)

显然，循环进程不可能少于 0 次重复下就终止，故若 T(n, x)有定义，则 T(0, x)为假，这样 S(0)为真。对循环进程证明的结果是∀n.S(n)，也就是

$$\forall n \bullet (T(n, x) \Rightarrow R)$$

由于 n 是不在 R 中出现的变量，故上式等价于

$$(\exists\ n \bullet T(n, x)) \Rightarrow R$$

∃ n • T(n, x)是说明循环进程在有穷步迭代后终止的前置条件，故不可能有更强于该式的规约描述了。

最后，我们必须证明循环体满足规约。由于定义循环的递归方程涉及条件进程，故这个证明分为两部分。这样，我们可以得到一般法则

L5 若¬T(0, x), 且 T(n, x) ⇒ ℳb

并有 SKIP sat (¬b ⇒ (T(n, x) ⇒ R))

和(X sat T (n, x) ⇒ R)) ⇒ ((Q; X) sat (b ⇒ (T (n + 1, x) ⇒ R)))

则(b * Q) sat ((∃ n • T (n, x)) ⇒R)

X11 我们证明用重复减法实现除法(5.5 节 X5)是满足规约 DIV 的。该任务很自然地分成两部分。第二部分也是更困难的部分是证明循环体满足某个适当的行为，即证明

$$(r \geq y) * (q := q+1; r := r - y) \text{ sat } (y > 0 \Rightarrow \text{DIVLOOP})$$

首先，我们需列出循环在 n 次重复前终止的条件

$$T(n) = r < n \times y$$

T(0)显然为假；命题∃ n • T(n)等价于 y＞0。这是表示循环终止的前置条件。证明的其余部分已在 X7 和 X5 中完成。读者可补全证明作为练习。 □

本节中的法则可作为纯顺序程序完全正确性的演算，在这些程序中不出现输入和输出算子。若 Q 是这样的一个纯顺序进程，则证明

$$Q \text{ sat } (P(x) \wedge tr \neq <> \Rightarrow <\surd> \wedge R(x, x^{\surd})) \qquad (1)$$

就说明了，若 Q 启动时，其变量的初始值使 P(x)，$R(x, x^{\surd})$就是 Cliff Jones 所定义的前/后置条件组。若 $R(x^{\surd})$不提及初值 x，断言(1)就等价于

$$P(x) \Rightarrow wp(Q,R(x))$$

这里 wp 是 Dijkstra 的最弱前置条件。

考虑非通信程序时，本章中的证明方法和已经熟知的一些方法，数学上是等价的，但是用了“tr=< >”和“tr=<√>”，这就使得符合系统比较笨拙。但当把这种方法扩

充到处理通信顺序进程时，这种额外负担是必要的，会比较容易接受这种符合系统了。

5.5.3 实现(Implementation)

顺序进程的初始和终止状态可用变量名至其值的函数表示。顺序进程则是由初始状态至后续行为的函数。成功终止√则表示为原子动作"SUCCESS。终止的进程就接受这个符号，并将进程映像为它所拥有的变量的终止值，而不是另一个其他进程。进程 SKIP 以初始状态为参数，以"SUCCESS 作为仅有的动作，将初始状态作为终止状态，故

SKIP = λs • λy • if y ≠ "SUCCESS then "BLEEP else s

赋值进程与其相似，但需要改变终止状态值，有

```
assign(x, e) =λs • λ y • if y≠ "SUCCESS then
                    "BLEEP
               else
                    update(s, x, e)
```

其中

update(s, x, e) == λ y • if y x then eval(e, s) else s(y)

eval(s, e) 是在状态 s 中，计算表达式 e 所得的结果。

我们不讨论 e 在状态 s 中无定义的情形。为简单起见，我们只实现单个赋值，多重赋值稍微复杂些，这里不做讨论。

在实现顺序进程组合时，必须先检验第一个运算对象，看其是否终止。若终止，则其终止状态传于第二运算对象。若不终止，则由第一个运算对象完成第一个动作

```
sequence(P, Q) = λs • if P(s)( "SUCCESS)≠ "BLEEP then
                         Q(P(s)("SUCCESS))
                    else
                         λy • if   = "BLEEP then
                                        "BLEEP
                                   else
                                        sequence(P(s) (y), Q)
```

条件进程的实现为

condition(P, b, Q) = λs • if eval(b, s) then P (s) else Q (s)

循环进程(b*Q)的实现留作练习。

上述的 sequence 的定义比 5.3.3 节中的定义更复杂一些，它以状态 s 作为第一个参量，而且它还需向它的运算对象提供状态参量值。这样前面各章中算子的定义都应作相应修改，都会变得更为复杂。简单的方法是用从属进程来模拟变量；但这个方法与使用传统的随机存取存储的方法，效率大为降低。另外，除了考虑数学处理的方便外，还应考虑效率；引入可赋值的程序变量作为一个新的基本概念，比之用以前介绍过的概念来定义程序变量，会更合理些。

第六章　共享资源(SHARED RESOURCES)

6.1 引言(Introduction)

在 4.5 节中我们介绍了具有命名的从属进程(m:R)[245]，它的任务就是满足单个主进程 S 的调用需求；为此，我们使用符号记法

(m:R // S)

假设 S 包含了或者恰好是两个并发进程(P ‖ Q)，而且 P 和 Q 两者都要求同一个从属进程(m:R)为它们服务。那末，P 和 Q 就不可能使用同样的通道和(m:R)通信；因为由‖的定义，P 和 Q 使用它们字母表中的公共通道和(m:R)通信时，要求 P 和 Q 两者同时接收或者发送相同的消息，但这种并行算子的强同步和我们所要的相距甚远(参见 4.5 节 X6)。我们所要的是，P 与(m:R)及 Q 与(m:R)间的某种交互式的通信，而不是它们三者的同时参予。(m:R)就象是 P 和 Q 的共享资源；每个进程独立地使用(m:R)，它们和(m:R)的交互作用是穿插进行的[246]。

如果事先知道哪些是共享进程，那就可以安排每个共享进程使用不同的通道与共享资源通信。就餐哲学家的故事(2.5 节)就使用这种技术：每把叉子由其相邻的两个哲学家共享，而男仆由他们五人共享。4.5 节 X6 也是一个这样的例子，由两个进程共享一个缓存，一个只使用它的左通道，而另一个只使用其右通道。

一般的共享方法可以通过多重标记(见 2.6.4 节)，它可有效地产生足够多的不同通道，每个通道用于一个共享进程和资源的独立通信。这些独立通道的通信动作可任意交互进行，不需要任何强制性的同步。但这种方法要求事先知道所有共享进程的命名；因此不太适用于从属进程，因为其主进程可能是由任意多个并发子进程组成的。本章要介绍一种适用于事先不必知道共享进程的个数和名称的资源共享技术。我们通过操作系统的例子来做相关的举例。

6.2 交叉式共享(Sharing by Interleaving)

6.1 节中的问题是因为并行算子‖对进程的并发同步行为的要求；为避免这个问题，可使用交叉式的并发性(P ||| Q)。其中 P 和 Q 有相同的字母表，它们和外部(共享

[245] 可以理解为子进程。

[246] 例如，可以是分时系统，Round Robin 的一种方式。

的)进程的通信则是任意穿插的。当然，这个算子不允许 P 和 Q 之间的直接通信[247]；但它们之间的间接通信是可以的，例如，可以通过设计适当的共享的从属进程来实现，可参见 4.5 节 X6 和下面的 X2.

例子

X1 子程序共享

doub : DOUBLE // (P ||| Q)

此处的 P 和 Q 都可调用从属进程，如包含有

(doub.left!v → doub.right?x → SKIP) □

尽管 P 及 Q 与从属进程的通信可以任意次序的穿插，但不会出现其中一个进程读到的是另一个进程希望得到的回答。为保证能免除这种危险，主进程中各子进程和共享的从属进程在左、右通道上的通信，需要严格地遵循交替出现的规则。因为这个原因，值得引进一种特殊的记号，以保证这样的交互遵从这个规则。这种记号和高级语言中传统的过程调用记号很想象，只是值参前冠以"!"，而结果返回值前冠于"?"。即令

doub!x?y = (doub.left!x → doub.right?y → SKIP)[248]

下面的一系列代数变换解释交互式共享的效果。当两个共享进程同时使用共享的子程序时，匹配的通信(调用，返回)以任意次序穿插出现，但一个进程的一对通信是不会被另一个进程的通信分割开的。为方便起见，我们使用下列简写，

在主进程中

D!v 表示 d.left!v

D?x 表示 d.right?x

在从属进程中

!v 表示 right!v

?x 表示 left?x

现在假设我们定义进程的行为如下：

D = ?x → !(x + x) → D[249]

P = d!3 → !D?y → P(y)

Q = d!4 → d?z → Q(z)

R = (d : D // (P ||| Q)) (同于 X1)

---

[247] 参阅 3.6 关于 Interleaving 算子的定义。P ||| Q 里 P 与 Q 本身之间不通信。算子|||刻画的是 P 和 Q 与环境之间的一种通信方式。

[248] 调用进程，或者主进程里 P 或者 Q 的行为。doub!x?y 表达了一次完整的调用，返回的过程。

[249] 子程序。

那么

p |||Q

= d! 3 → (d！ 4 → (d？ y → P(y) |||Q)

d!4 → (P |||(d?z → Q(z)))　　由 3.6.1 节 L7

每个共享进程都向共享的进程输出消息。至于接受哪个消息，则由共享的进程决定。由于它愿意接受两者之任一，故屏蔽这一选择动作后，就出现了非确定性。

(d : D // (P |||Q))

= ((d : (!3 + 3 → D)) // ((d?y → P(y)) |||Q))

⊓

((d :(！ 4 + 4 → D)) // (P |||(d?z → Q(z))))

= (d : D // (P(6) |||Q)) ⊓ (d : D // (P |||(Q(8)))

上述推理用到 4.5.1 节 L1，3，5.1 节 L5 等。共享的进程将计算结果提供给已准备接收结果的共享进程。由于两个共享进程之一仍在等待输出，只有提供参量的进程才准备接收结果。从而可见，在调用共享的子程序过程中，输出和输入动作必须严格地交替出现。

X2 数据结构共享

在预订机票系统中，订票工作是由很多售票员承担的，他们的订票工作穿插进行的。每订一张票，就在航班旅客表中增加一名乘客，并且能告诉本乘客是否已经订过票，以免重订。对这个大为简化了的例子，可用 4.5 节 X8 中实现的集合，作为共享的从属进程，并冠以航班号为进程名。

AG109 : SET //(... (CLERK|||CLERK |||...)...)

每个 CLERK 每为一名乘客订一张票，就调用一次

AGI 09!pass no?x

这代表(AG109.1eft!pass no → aG109.right?x → SKIP)　　□

在这两个例子中，每使用一次共享资源包括两次通信，一次发送参数，一次接收结果；在每对通信后，从属进程又恢复为可为两个进程中任一进程服务。我们有时要求两个进程间能实现批处理的通信，不希望第三个进程打断干预。例如，在很多并发进程间共享一台非常昂贵的打印机。而每次使用时，要求连续地输出一个多行组成的文件，在输出中，不会夹杂另一个进程输出的行。为了实现这一点，在输出打印

前，必须先实现一个获取设备的事件，由此得到独占资源的权利；在完成输出后，为了使资源可供其它进程使用，必须释放资源[250]。

X3 行式打印机共享

LP = acquire → μX • (left?x → h!s → X ｜ release → LP)

此处，h 是将 LP 连至行式打印机硬件的通道。在获取事件后，进程 LP 不断地从左通道向硬件成行地拷贝数据，但遇到释放信号就恢复到最初的状态，再为其它进程服务。进程 LP 用作共享资源，可出现于[251]

lp：LP // ...(P ||| Q)...

在 P 或 Q 中，组成同一文件的各行以 lp.acquire 和 lp.release 括起来

lp.acquire → ...lp.left!"A. JONES" → ...

lp.left!nextline → ...lp.release →… □

X4 X3 的一种改进

当行式打印机由多个用户共享时，在输出文件后，要将整条打印纸按文件裁开。为此，打印纸以页计，页间用一行透纸孔隔开；打印机的硬件可执行操作走页(throw)，这个操作使打印纸快速前进到当前页的页末——打印纸的下一条向外折缝。为帮助区分不同文件的输出，文件应从页头开始，结束于页末，而且在文件的最后一页的页末，和第一页的页头，都打印一行星号。为避免混淆，文件中不允许有整行的星号。故令

```
LP = (h!throw → h!asterisks →
        acquire h!asterisks →
                μX • (left?s → if s ≠ asterisks then
                                h!s → X
                                else
                                        X
                                ｜ release → LP))
```

使用这个 LP 的方法同于上例 X3。

上面的最后两例中使用了 acquire 和 release 来防止不同行间的任意穿插，同时也不会引致死锁。但当共享的资源多于一种时，这种方式的共享可能会有死锁的危险。

□

[250] 可以理解为操作系统中锁的机制。

[251] Hoare 2015 CSP 电子版中关于 lp 的定义没了。译者认为可能是遗漏。

X5 死锁

安妮和玛丽是好朋友，也都是好厨师；她们合用一口锅和一个炒勺，在她们需要时，就获取、使用和释放这两个共享资源。

UTENSIL (acquire → use → use → ... release → UTENSIL)

pot : UTENSIL // pan : UTENSIL // (ANN ||| MARY)

安妮按某种菜谱做菜，先用锅后用勺，而玛丽则先用勺后用锅，即

ANN = ... pot.acquire → ... pan.acquire → ...

MARY = ... pan.acquire → ... pot.acquire → ...

遗憾的是，她们决定同时做饭。她们都先获取各自的第一个用具；但当她们需用第二个用具时，都没法得到，因为各被对方占用着。

安妮和玛丽的故事可用二维图表示(见图 6.1)，图中安妮的活动由纵坐标表示，而玛丽的活动由横坐标表示。系统以左下角为起点，即作为她们的活动的开始处。安妮每完成一个动作，系统向上移一步。玛丽每完成一个动作，系统则向右移一步。图上的一条折线表示安妮和玛丽一系列穿插动作。这条折线直达图形的右上角，两位厨师都可享用她们的饭菜。

但不总是那样幸运的。她们不能同时使用同一个共享工具，因此折线无法通过某些长方形地区。例如，在有右上至左下影线的区域，两个厨师要合用炒勺，故不可通过。类似地，由于不能合用锅，故折线也不能通过有左上至右下影线的区域。折线一旦抵达这些区域，只能沿着垂直边向上，或者沿着水平边向右。这

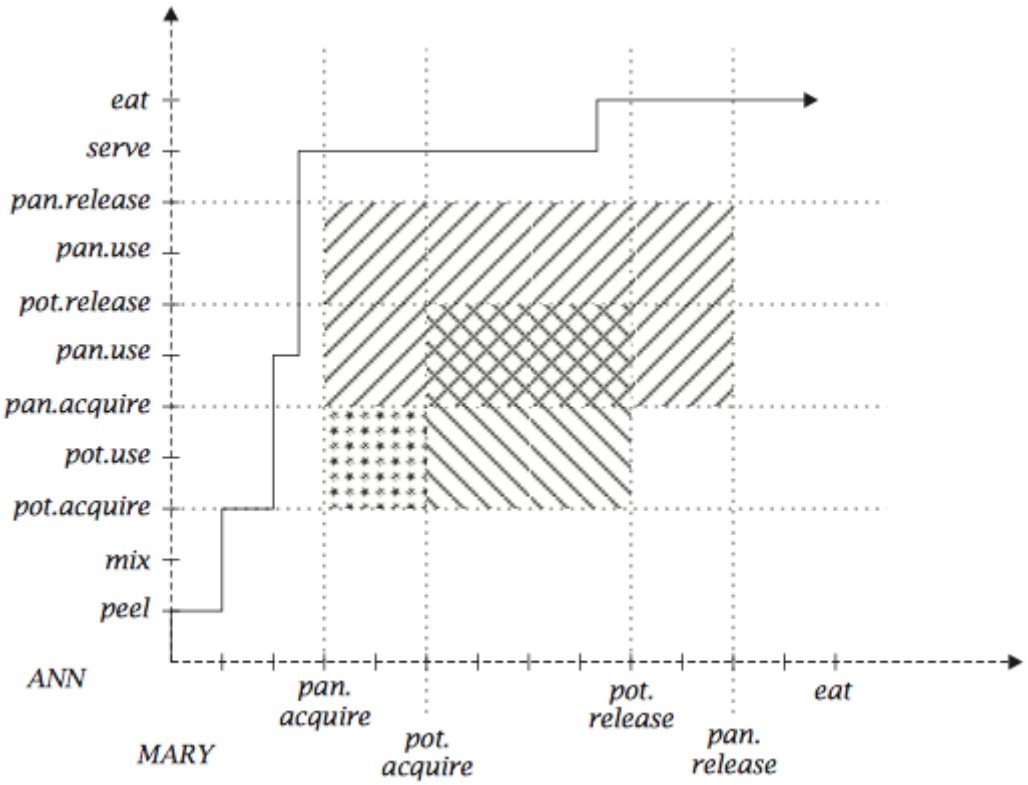


图 6.1

表示，其中的一位厨师等待另一位释放用具。

现在考察带有小点的区域。一旦折线进入这一区域，就不可避免地要在区域的右上角处死锁[252]。这个图形说明，死锁是由禁区的凹点造成的，而且是面向原点的凹点才会造成死锁。阻止死锁发生的唯一可靠的方法是扩大禁区，使其包含危险区，即消除凹点。另外一种方法是，增加一种人为的资源，在获取任一用具前，必须先获取这一资源；而在两个用具都释放后，才能释放这一资源[253]。这是一种免除死锁的方法，这种方法类似于哲学家就餐故事(2.5.3 节)中设置男仆的想法；可将就座权设想为一种资源，五个哲学家共享四个就座权。另一种更简单的方法是，规定使用两种用具的厨师，必须先获取炒勺[254]。本例出自 E. W. Dijkstra。 □

上面提出的简单方法，可推广到多用户和多资源情形。只要规定所有用户都按固定的相同次序获取资源，就可以避免死锁。用户还应在用完资源后立即释放，释放的次序是无关紧要的。用户也可不按规定次序获取资源，但要保证在获取某一资源

[252] Ann 拥有了 pot，想要 pan；Mary 有了 pan，想要 pot。发生死锁现象。
[253] 增加一个全局锁，有点类似 Big Lock。
[254] 通过顺序化的方法，避免乱序。

时，按标准次序在其后的所有资源都已释放。扫描用户进程的全文，就可确定资源的获取和释放是否遵循这类规定。

6.3 存储共享(Shared Storage)

本节的目的是说明为何要反对使用共享的存储区；如果你亦已信服这一论点，则可跳过不读。

并发进程系统的行为可在单个传统的程序内存计算机上实现，所用的方法称作分时技术；单处理机交替地执行各个进程，进程间的交替由外部设备的中断所引发，或由一个定时器触发。这种实现方法自然允许并发进程享用公共存储区，在各个进程中可以使用通常的机器指令存取共享的存储单元。

在我们的理论中，是用共享变量(4.2 节 X7)来模拟共享存储单元，例如

(count : VAR // (count.left!0(P ||| Q)))

共享存储不同于 5.5 节中的局部存储。有关顺序进程的推理的法则很简洁，其原因在于每个变量至多只能被一个进程更新[255]；允许不同进程对同一变量任意穿插地赋值，会带来很多危险，这些危险也是已有法则无法处理的。

这类危险可由下列例子说明

X1 干扰[256]

共享变量 count 用于记录某一重要事件的发生次数。每当该事件发生时，进程 P 或 Q 就用一组通信来更新计数变量

count.right?x; count.left!(x + 1)

不幸的是，这两个通信可能被另一个进程的类似的一对通信叉开，出现

count.right?x →

count.right?y →

count.left!(y + 1) →

count.left!(x + 1) →...

其结果是，计数变量的值只增加了 1，而未增加 2.这类错误就称为干扰，在设计共享存储的进程时，很容易犯这种错误。而且这种错误的出现是极其不确定的；很难重现，因此几乎不可能用传统的测试技术去诊断。我对一些已普遍使用的操作系统仍持怀疑态度，因为它们输出的统计数字、账单和一览表总不是很准确的。 □

[255] 这里指的是 5.5 节里的顺序进程对局部变量的处理方式。

[256] 指的是数据或者状态的不一致性问题。

解决的方法之一是，在不允许被穿插的一串操作执行的过程中，确保不出现进程的交替。这串操作称作临界区。在单处理机上实现这一方法时，往往要求在进入临界区后，禁止所有的中断，从而达到互斥的要求。这样做的缺点是大大延迟了对中断的响应；更坏的是，如果计算机上有两个处理部件，这个方法就完全不灵了[257]。

更好的解决办法是用 E. W. Dijkstra.的二值互斥信号量。一个信号量亦可看作是一个进程，它交替地执行 P 动作和 V 动作

SEM = (P → V → SEM)

信号量用作共享的资源

(mutex •. SEM//...)

在进入临界区时，进程必须发出信号

mutex.P

而在退出临界区时，必须执行事件

mutex.V

计数器增值的临界区就应写成

```
mutex.p →
        count.right?x → count.left!(x + 1) →
                mutex.V → ...
```

只要所有进程都遵从这个规定，就不会出现两个进程彼此干扰对计数器的更新。

但若有进程不执行 P 或 V，或执行的次序有误，则会出现乱七八糟的情形，有时甚而无法捉摸。

阻止不一致性干扰的更强有力的方法是将保护机制植入共享存储的设计中，充分利用存储的使用方式。例如，用于计数的变量的增值操作就应是单个原子操作

count.up

而共享资源就可设计为 CTo (1.1.4X2)

count : CTo // (... P ||| Q)

事实上，我们有许多很好的论据表明，共享的资源应该通过专门设计，并发系统不应该采纳通常的纯共享存储。这样，不仅可以避免出现不一致性问题的干扰，而且使你的设计既可以用分布式处理部件的网络实现，也可以在带有共享存储装置的单处理机或者多处理机上有效地实现。

[257] Hoare 的著作成文于 1970-80 年代。现代操作系统的演变已经可以很好的支持各种粒度的锁，多处理器系统了。

6.4 多重资源(Multiple Resources)

6.2 节中我们讨论了，如何使多个行为不同的进程共享单一的从属进程。为了保证任何时刻至多只有一个进程占用资源，每个共享进程必须遵从某些纪律，如交替地进行匹配的输出和输入，或者交替地得到获取和释放控制信号。这类资源称为串行化重用资源(serially reusable)。本节中，我们介绍如何使用进程数组表示行为相同的多重资源。由数组的下标来确保已获取该资源的进程可安全地与其通信。

我们将大量使用下标和带下标的算子，它们的含义是不言自明的。例如

$\|_{i<12} P_i = (P_0 \| P_1 \| ... \| P_{11})$

$|||_{i<4} P = (P \,|||\, P \,|||\, P \,|||\, P)$

$\|P_i = (P_0 \| P_1 \| ...)$

$\Box_{i\geq 0}(f(i) \rightarrow P_i) = (f(0) \rightarrow P_0 \mid f(1) \rightarrow P_1 \mid ...)$

f 是 1-1 函数，故分支的选择完全由环境决定。

例子

X1 重入子程序

共享的子程序是串行可重用的，即每个时刻只能被一个进程所调用。如果子程序的计算量较大，则会相应地阻滞调用进程。在多处理器的情况下，应允许使用多个子程序实例，在不同处理器上并发地运行。可有多个并发实例的子程序叫作可重入子程序，可用并发进程数组来定义，如

doub : $(\|_{i<27}$: DOUBLE)) //...

对这个子程序的典型的调用可以是

(doub.3.left!30 → doub.3.right?y → SKIP)

由于使用了指标 3，故可确保所得到的调用的结果值是来自接受参数值的 doub 的实例；即使同时还有另外的并发进程在调用另外的实例，也不会形成干扰。消息间的可能穿插如

doub.3.left.30, ... doub.2.left.20,

.. doub.3.right.60, ... doub.2.right.40, . . .

当进程调用重入子程序时，数组的哪个元素响应这个调用是无关紧要的；只要是没被占用都可响应。故在调用进程中不应规定特定的指标，如 2 或 3，而应使用下列结构，从而可任意选择指标

$\square_{i\geq 0}$ (doub.i.left!30 → doub.i.right?y → SKIP)

这个结构遵从重要的规定，即发送参数值时所用的下标和以后接收结果时所用的下标是相同的。 □

在上例中，子程序的同时动作的实例数不超过 27,。但很容易让一个处理器同时处理更多个进程，因之需要引入并发进程的无穷数组以避免给实例个数以任意限制。如

doub : ($\|_{i\geq 0}$ i : D)

这里的 D 可以只用于一次调用，用后就停止，即

D = left?x → Right!(x + x) → STOP

重入次数不作限制的子程序叫作过程。

每个过程调用

$\square_{i\geq 0}$ (doub.i.left!x → doub.i.right?y → SKIP)

的效果，应该和调用从属进程 D

(doub : D // (doub.left!x → doub.right?y → SKIP))

的效果完全相同。但后者是一个本地过程调用，即过程和调用进程在同一个处理器内执行；而共享的过程调用称为远程调用，即过程可能在一个远处的处理器上执行。远程和本地调用的效果相同，使用远程调用往往出自政治的或经济的原因，例如，为保证过程软件的秘密，或者在带有昂贵的专用设备的机器上才能执行过程等。典型的贵重设备如大容量的备份存储器，例如，磁盘或磁泡存储等，由于太昂贵而无法在运行调用进程的机器上设置。

X2 备份存储的共享

一个存储介质分为 B 个扇区，各扇区可被独立地读写。每个扇区贮存一个信息块，由左边输入，向右边输出。遗憾的是，这个存储介质用的是读出破坏技术，因此写入的信息块只能读出一次。这样，每个扇区就如同一个 COPY 进程(4.2 节 X1)，而不象 VAR 进程(4.2 节 X7)。整个备份存储器就是扇区的数组，其下标值小于 B，即

BSTORE = $\|_{i<B}$ i : COPY

这个备份存储可用作从属进程

(back : BSTORE //...)

在主进程中，通过通信

back.i.left?bl → ... back.i.right?y → ...

...

来使用该存储器。

备份存储亦可被并发进程所共享。这种情况下，动作

$\square_{i<_B}$ (back.i.left!bl → ...)

可以同时获取任意一个(编号为 i)的可用的扇区，并同时写入值 bl。类似地，back.i.right?x 作为单个动作，能够同时完成读出扇区 i 的内容，并释放该扇区以备其它进程使用[258]。这种简单的使用方式促使我们可以通过 COPY 进程模拟扇区的行为；至于读出破坏的说法只是一种借口而已。 □

为顺利地共享备份存储，要求共享进程遵守严格的规定。只有一个进程向扇区输出了一个信息块未被破坏前，该进程才能从该扇区读走这个信息，而且每个这样的备份，读取恢复行为，必须配对。不遵从这类规定，会导致死锁，甚至出现更为糟糕的混乱局面。不需付出很多代价，就可对共享进程加之上述约束；在介绍完下面例子后，我们就介绍这种方法，并将在 6.5 节的操作系统的模块设计中详尽解释。

X3 两台行式打印机

两台一模一样的行式打印机为若干个用户进程服务。每台打印机都需要保护，以免打印过程中不同文件间的数据相互穿插，6.2 节 X4 中的 LP 提供了这种保护。我们就使用 LP 的两个实例所组成的数组来表示这两台打印机，每个标以一个自然数，指出它在数组中的位置，既有

LP2 = (0 : LP ‖ 1 : LP)

数组本身亦给以名字，这样可用作共享的资源

(lp2 : LP2// ...)

LP 的每个实例现有两个前缀，一个是名字，一个是指标；和用户进程的通信由三个或四个成分构成，例如

lp. 0.acquire, lp.i.left. "A.JONES", ..

如同调用重入过程，当进程需要获取资源数组中的一个资源时，选中哪一个数组元素是无碍的。所有能响应获取信号的元素都是可接受的。可用一般的选择结构来表示这种任意的选择，如

$\square_{i\geq 0}$ (lp.i.acquire → .... lp.i.left!x → ...; lp.i.release → SKIP)

[258]定义为一个原子操作，不可分割。例如，被打断，或者中间插入其它动作。

第一个 lp.i.acquire 为获取两个 LP 进程中的能响应该事件的任一进程。若没有响应，则等待；若两者都能响应，则以非确定的方式选一个。获取一个资源后，约束变量 i 以所选的资源的下标为其值，然后以后的通信就能正确无误地指向这个资源。

当一个进程已获取某共享资源，该资源就等同于该进程本地的从属进程一样，只和该进程通信。我们现在修改一下从属进程的符合记法，并将复杂的结构

$\square_{i\geq 0}$ (lp.i.acquire → ...; lp.i.left!x...; lp.i.release → SKIP)

改写成

(myfile :: lp// ... myfile.left!x...)

其中，局部名 myfile 表示带有下标的名字 lp.i，而术语获取和释放等也被删去了。新记号"::"叫作远程从属关系；它和原来的记号":"不同，在它的右边不是一个完整的进程，而是一个远程进程数组的名字。 □

X4 两个输出文件

一个用户进程同时使用两台行式打印机，同时输出两个文件 $f1$ 和 $f2$，写成

($f1$ :: lp // ($f2$ :: lp // ... $f1$.left!sl → $f2$.left!s2 → ...))

用户进程穿插地输出两个文件中成行的数据；而且每一行都在相应的打印机上打印。若用户同时要用三台打印机，这将无疑地会导致死锁，因为这里只有两台打印机可供使用。类似地，若两个并发进程都要使用两台打印机，也会导致死锁，这和安妮和玛丽的故事(6.2 节 X5)是一样的。 □

X5 临时文件

临时文件用于输出一串信息块。输完后，文件就反卷，整串信息块又从头读一遍。读完后，就给出一个 empty 信号；然后就不能再读写了。临时文件很象向磁带上输出的文件，在读以前必须反卷。信号 empty 如同记号 eof(文件末)，上述行为可以表示为

SCRATCH = $WRITE_{<>}$

$WRITE_s$ = (left?x → $WRITE_{s^\wedge<x>}$

| rewind → $READ_s$)

$READ_{<x>^\wedge s}$ = (right!x → $READ_s$)

$READ_{<>}$ = (empty → $READ_{<>}$)

临时文件亦可很方便的用作非共享的从属进程，如

(myfile : SCRATCH // ... mfile.left!v ... myfile.rewind ...

.... (myfile.right?x → …

| myfile.empty →...)...)

后面将用此作为一个共享的进程的模型。 □

X6 备份存储中的临时文件

X5 中的临时文件可以很容易的通过计算机主存中的一串信息块来实现。但若这些快很大，串也很长，存在主存中很不经济了，最好存到备份存储装置中。因为临时文件中的信息块只需要读写一遍，故可采用 X2 中的读出后数据丢失的备份存储器。在主存中存放一个不大的临时文件，存放着备份存储器中贮存相应信息块的扇区的索引；这样就可按所要求次序将信息块再正确地读入主存。这个行为可表示为

BSCRATCH = (pagetable : SCRATCH //
μX • (left?x ($\Box_{i<B}$ back.i.left!x → pagetable.left!i → X)
| rewind →
pagetable.rewind →
μY • (pagetable.right?i →
back.i.right?x → right!x → Y
| pagetable.empty → empty → Y)))

BSCRATCH 使用名称 back 命名备份存储器(X2)，这样备份存储器就成了它的从属进程。为提供这样的从属进程，令

SCRATCHB = (back : BSTORE // BSCRATCH)

SCRATCHB 子进程的唯一不同处是规定了最多只可以有 B 个信息块。 □

X7 串行重用临时文件

设想在多个穿插使用的用户中共享备份存储器中的临时文件，用户象使用共享的行式打印机(6.2 节 X3)那样，采用获取、使用和释放的动作。这样，必须修改 BSCRATCH，使其能接受获取和释放信号。若用户在未读完文件前，就提前释放他的文件，那就使得备份存储中未读出的信息块的扇区不能再使用了。为消除这种危险，需设计一个循环程序，读出这些信息块，并把它们删去。这个程序是

SCAN = μX • (pagetable.right?i →
back.i.right?x →
X
| pagetable.empty → SKIP)

共享的临时文件先获取用户，然后如 BSCRATCH 般动作，释放信号引起中断(5.4 节)转至 SCAN 进程，即

SHBSCR = acquire → (BSCRATCH^(release →SCAN))

串行重用的临时程序既是下列循环

*SHBSCR

它以 BSTORE 作为其从属进程

back : BSTORE // *SHBSCR □

X8 多重临时文件

上两例中，每个时刻只用一个临时文件。备份存储通常是非常大的，足可以容纳多个临时文件，它们占据着互不相重的一些扇区。备份存储区故可由许多临时文件的无所共享(共存)。每个临时文件在输出信息块时，先获得一个扇区，而在读入这个信息块时进入主存后，自动地释放该扇区。为共享备份存储，可用多重标记的技术(2.6.4节)，将构造共享进程数组时的下标(自然数)作为标记。令

FILESYS = N : (back : BSTORE) // ($\|_{i\geq 0}$ i : SHBSCR)

此处 N ={ i | i ≥0 }。

这个文件系统可用作从属进程，在任意多个用户中穿插地共享。可写为

filesys : FILESYS //... (USER1 ||| USER2 ||| ...)

在每个用户进程中，可用远程从属关系获取，使用和释放一个新的临时文件，即

myfile ::filesys // (... myfile.left!v ... myfle.rewid ... myfile.right?x ...)

若不考虑资源的限量，上述式子等效于 X5 中独属的临时文件

(myfile : SCRATCH //

... myfile.left!v ... myfile.rewind ... myfile.right?x ...) □

X8 的文件系统结构和使用方式是在不定个数用户中共享有限的实际物理资源(备份存储器的扇区)的一个范例。用户并不与物理资源直接通信；用户通过和对接一个中间的的虚拟资源(SHBSCR)，把它作为自己的从属进程一样使用。虚拟资源的作用有二个方面。

(1) 它提供给用户一个良好的接口；如 SHBSCR 将散在备份存储器中的扇区收集起来，形成单一的连续的临时文件。

(2) 它保证对实际物理资源的使用是符合规定的，如 SHBSCR 确保每个用户仅从分配给他的扇区读入信息，而且在使用后，不会忘记释放扇区。

由(1)又保证(2)中的规定是不必付出很大代价就能实现的。

实际物理资源和虚拟资源的上述典型用法，在设计资源共享系统中是非常重要的。这种模式的数学定义相当复杂，它使用自然数的无界集来实现虚拟进程的动态建立，以

及和这些虚拟进程通信的新通道的动态建立。在计算机上实际实现时，可用控制块及指向有效的数据块的指针等表示。为了高效地应用这种方法，最好先不讨论实现问题。但愿意更深入了解的读者，下面关于 X8 的深入解释应该有所帮助。

在一个用户处理器内，临时文件是用远程从属关系建立的，即

myfile :: filesys // (... myfile.left!v ... myfile.rewind ... myfile.right?x ...)

按照远程从属关系的定义，上式右方等价于

($\square_i \geq 0$ filesys.i.acquire →

filesys.i.left!v ... filesys.i.rewind ... filesys.i.right?x …

filesys.i.release → SKIP)

在 filesys 及其用户间的全部通信是以 filesys.i 开始，其中 i 是 SHBSCR 的某个实例的下标，这个实例是由某个用户在某种场合下已获取的。另外，对这个实例的使用过程都以一对信号围起，确保数据一致性。即

(filesys.i.acquire → ... filesys.i.release)

在从属进程一方，每个虚拟的临时文件以获取用户为起始，继之按照 X5 和 X6 中说明的格式展开

(acquire → ...left?x ... rewind ... right!v ... release...)

虚拟临时文件亦和从属的 BSTORE 进程通信，但向用户屏蔽。每个虚拟临时文件的实例都有不同的下标 i，而且带有名称 filesys。故每个案例的外部可见的行为是

(filesys.i.acquire →

filesys.i.left?x ... filesys.i.rewind ... filesys.i.right!v …

filesys.i. release)

这恰好和上一段落中用户的通信格式相匹配。匹配的信号对 acquire 和 release 保证了在用户已获取的临时文件中不会遇到其它用户的干扰。

我们现在考查 FILESYS 中的虚拟临时文件和备份存储间的通信。这些通信已对用户屏蔽，甚至不带有名字 filesys。有关的事件是

i.back.j.left.v 表示有临时文件数组的第 i 个元素向备份存储的

第 j 个扇区传送信息块 v

i.back.j.right.v 表示反向的一个通信

备份存储的每个扇区的行为如同进程 COPY。在扇区上加以序号 j 和名字 back 后，其行为如

μX • (back.j.left?x → back.j.right!x → X)

用自然数作为多重标记，其行为为

$$\mu X \bullet (\Box_{i \geq 0}\, i.back.j.left?x \rightarrow (\Box_{k \geq 0}\, k.\ back.j.right!x \rightarrow X))$$

这样，第 j 个扇区可以和虚拟临时文件数组的任意元素通信了。每个虚拟文件只从已被其写入的扇区中读出信息块。

上述说明中，自然数 i 和 j 的作用是允许任意临时文件和磁盘上的任意扇区通信，而且也可安全地与已经获取该扇区读写权限的用户通信。

这些索引也是电话交换机中的交错开关的一种数学刻画，这种开关装置使任意两部电话机之间的通信成为现实。图 6.2 是这个通信过程的一张略图。

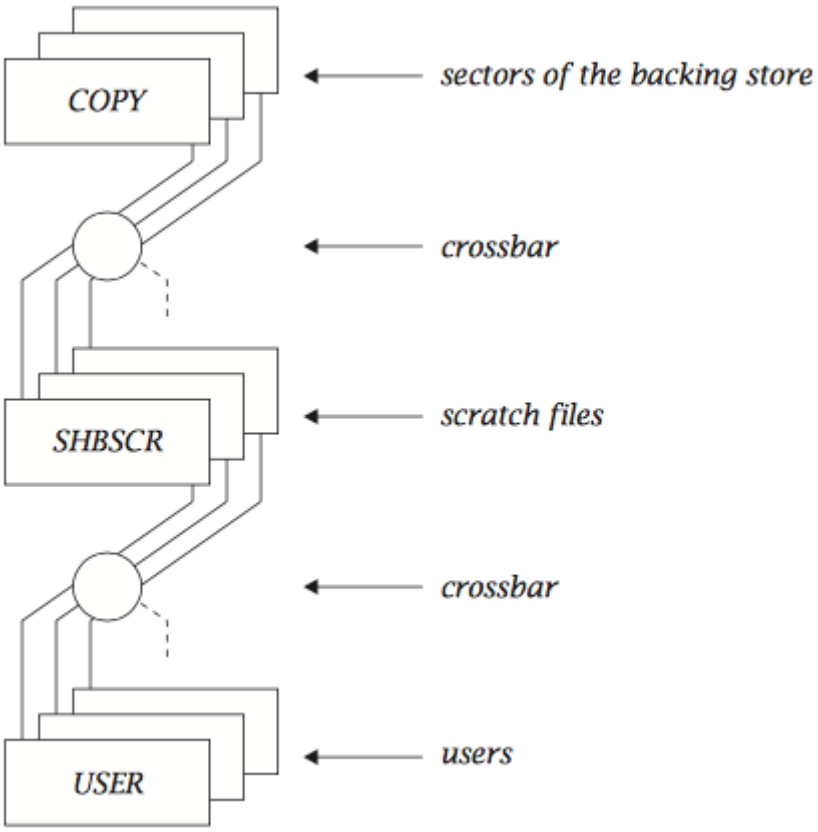


图 6.2

若备份存储中有无穷多个扇区，FILESYS 的行为和有类似结构的临时文件数组完全一样。这个数组是

$$\|_{i \geq 0}\, i : (\text{acquire} \rightarrow (\text{SCRATCH} \triangle \text{release} \rightarrow \text{STOP}))$$

备份存储量有限时，当备份存储已满，而所有用户还在向临时文件写入信息，则会出现死锁的危险。实用中，当后备存储将满时，就会推迟新文件的获取，从而极大地减少了出现这类危险的可能性。

6.5 操作系统(Operating System)

在使用一个大型计算机时，用户将他们的程序放在读卡机上，由计算机读入执行。程序所用的数据也紧随在程序之后输入。批处理操作系统的任务是负责这些作业共享计算机的资源。我们假定用户程序由 JOB 进程负责运行，它从通道 cr.right 上输入程序卡，并运行程序从读卡机上读入数据和加工数据，再将加工结果由通道 lp.left 输出。我们不必了解 JOB 的内部结构；在早期，这是一个 FORTRAN 的监控系统。我们假定在启动这个进程后的合理的时间内，它会成功终止。JOB 的字母表定义为

αJOB = {cr.right, Ip.left, √}

令 LPH 表示行式打印机，GRH 表示读卡机，则单用户单作业可由下列进程执行

JOB1 = (cr : CRM // lp : LPH // JOB)

只运行一个作业，然后就终止的操作系统是没有多大用处的。共享一个计算机的最简单的方法是串行地运行作业，一个接着一个，即

BATCH0= (cr : CRH // lp : LPH // *JOB)[259]

这个设计忽略了很多重要的管理细节，如不同作业输出的文件应该分开，卡片仓内的作业卡片亦应该分开，以免一个作业会读入下一个作业的卡片。为解决这些问题，我们用 6.2 节 X4 定义的 LP 进程，及下面 X1 中的 CR 进程，定义

JOBS = *((cr.acquire → lp.acquire → JOB) ;
(cr.release → lp.release → SKIP))

BATCH1= (cr :CR // lp : LP // JOBS)

BATCH1 是一个最简单的可使用的操作系统的抽象描述，可以使得很多用户共享一个计算机，用户的作业逐个顺序执行。操作系统的目的是加速作业之间的过度，并且保证每个作业不受其它作业的干扰。

例子

X1 一个共享的读卡机

[259] Hoare 2015 CSP 电子版里，此处 BATCH0 定义漏了 JOB 的循环*符号。

有一种专用的分割卡，插在每个作业文件的最前面。读卡机需要读入一个作业文件的所有卡片，然后才能释放。如果超出分割卡后，用户若还想读入卡片，能得到的只是分割卡的拷贝。若用户尚未读到分割卡就释放读卡机，则全部剩余的卡片都要跳过。多余的分割卡会被跳过。从硬件读入信息是用输入 h?x。

共享的读卡机需超前读一张卡片，从而缓存的卡片作为一个索引。

$CR = h?x \rightarrow \text{if } x = \text{separator then } CR \text{ else } (\text{acquire} \rightarrow cR_X)$

$CR_X = (\text{right!}x \rightarrow h?y \rightarrow$

if $y \neq$ separator then

CRy

else $\mu X \bullet (\text{right!separator} \rightarrow X \mid \text{release} \rightarrow CR)$

$\mid$ release $\rightarrow$

$\mu X \bullet (h?y \rightarrow \text{if } y = \text{separator then } CR \text{ else } X))$

跳过最初的一串分割卡后，进程获取它的用户，并向其右通道拷贝读入非分割卡。读到分割卡后，重复地传送这个分割，通知用户释放读卡机。若用户提前释放了读卡机，卡片仓中下个分割卡前的剩余卡片继续被读入并删除。

操作系统 BATCH1 在逻辑上是完整的。但是因为中央处理器的速度比较快，远胜于读卡机和打印机的有关作业的输入和输出的速度。为了使输入，输出和处理速度匹配，就需要使用多个读卡机和打印机。因为每个时刻中央处理器只能加工一个作业，多余的读卡机可以提前读入下面一个或多个作业文件；多余的打印机可输出前面的一个或多个作业的结果。在被 JOB 使用前，每个输入的文件必须暂存于某个临时文件中；而每个输出的文件，从由 JOB 产生至打印机上实际输出的间隔内，也必须暂存起来。这种技术叫作 Spooling 技术[260]。

一个支持 Spooling 技术的操作系统的总体结构是

OPSYS1 = insys : INSPOOL //

outsys : OUTSPOOL//

BATCH

这里的 BATCH 类似于 BATCH1，但它使用远程从属关系来获取任意一个正等待输入的文件，和要打印输出的文件，即

BATCH = * (cr :: insys// lp :: outsys // JOB)

INSPOOL 和 OUTSPOOL 的定义见下面的两个例子。

[260] Spooling 的缩写为 Simultaneous Peripheral Operating On Line。中文通常称为假脱机技术。

X2 缓冲脱机输出

每个用户进程使用一个虚拟打印机，这个虚拟打印机用一个临时文件(6.4 节 X5)暂存用户进程输出的信息块。用户进程释放虚拟打印机后，实际的打印机(6.4 节 X3)就被获取来输出暂存文件中的内容。故

VLP = (temp : SCRATCH //
μX • left?x → temp.left!x → X
|release → temp. Rewind →
(actua l:: lp //
μY • (temp.right?y → actual.left!y → Y
| temp.empty → SKIP)))

所需的虚拟行式打印机的无界数组可定义为

VIPS= $\|_{i\geq 0}$ i : (acquire →VIP)

实际的行式打印机(6.4 节 X3)可用多重标记，作为 Spooling 操作系统的本地资源，由 VIPS 数组的所有元素共享(见 6.4 节 X8)，即

OUTSPOOL = (N : (lp : LP2) // VLPS) □

X3 缓冲脱机输入

输入缓冲机制类似于输出缓冲机制，区别在于只是需要预先获取一个真的读卡机，而在输入一个作业完成后释放这个读卡机；然后一个用户进程获取执行作业时，就把卡片的内容输出给这个进程。

VCR = temp : SCRATCH //
(actual :: cr //
(μX • actual.right?x → if x= separator then
SKIP
else
temp.left!x → X));
(temp.rewind → acquire →
(μY • (temp.right?x → right!x → Y
| temp.empty → right!separator → Y)) △
(release → SKIP))

INSPOOL = (N: cr : (0 : CR ‖1 : CR)) // ($\|_{i\geq 0}$ i : VCR)

输入和输出脱机缓冲机制可以为 JOB 进程提供无限多个虚拟读卡机和虚拟行式打印机。这样，就可同时运行多个 JOB 进程，由它们共享这些虚拟资源。由于这些作

业间互不通信，简单的轮换穿插方式就可以适用于这种资源共享。这种技术称为多道程序设计；如果使用多个处理器硬件，就称为多处理机系统。多道程序设计和多处理机系统的逻辑功能是相同的。确实，下面所定义的操作系统和 OPSYS1 有相同的逻辑描述

OPSYS = insys : INSPOOL // outsys : OUTSPOOL // BATCH4

此处

BATCH4 = $(|||_{i<4}$ BATCH)

从数学上看，由单道程序设计转向多道程序设计是相当简单的；但在历史上，多道程序设计曾带来很大麻烦。

OUTSPOOL(X2)设计中的 VIP 使用了 SCRATCH 进程，把它作为从属进程，贮存由每个 JOB 所生产的数据，然后输给实际打印机打印。一般情况下，输出文件都很庞大，不宜存在计算机主存中，应象 6.4 节 X8 中建议的，放在备份存储器内。而所有的暂存文件要共享同一个备份存储，故需将 VIP 中的从属进程

temp :: SCRATCH // ...

替换为远程从属进程

temp :: filesys // ...

而将 6.4 节 X8 中的文件系统作为输出卷宗的从属进程，有

(filesys : FILESYS // OUTSPOOL)

如果输入的卡片量也很大，那么对 INSPOOL 也应做相应修改。若有另外一个后备存储，则可用此单独贮存输入文件。否则，输入和输出缓冲的暂存文件就需要共享单一的后备存储。这就意味着，必须用多重标记，将 FILESYS 当作一个从属进程，由两个卷宗共享；这样就要涉及系统结构的变更。我们用自顶向下的方式重新设计这个系统，并尽可能多地使用前面已定义的模块。

这个操作系统由一个批处理的多道程序系统 BATCH4，和一个作为从属进程的输入输出系统所组成，即

OP =IOSYSTEM // BATCH4

输入输出系统中，输入和输出共享一个文件系统，即

IOSYSTEM = SH : (filesys : FILESYS) //

(lp : OUTSPOOL' ‖ cr : INSPOOL')

和 SH ={lp.i | i ≥0 } ∪ { cr.i | i ≥0 }而且 OUTSPOOL'和 INSPOOL'等同于 X2 及 X3，除了只是用远程从属进程

temp :: filesys

代替等价的

temp : SCRATCH

在上述四个操作系统(BATCH1, OPSYS1, OPSYS, 和 OP)的设计中，我们特别强调的是模块化的长处。由此，我们能够在后来的系统中，重用前面系统中的大部分模块。更重要的是，每个细节都只涉及系统中的一个或两个模块。因此，一旦要修改某一细节，就很容易确定需要修改的模块。例如，比较容易的改动有

- 行式打印机的个数
- 读卡机的个数
- 并发的作业批数

但并不是所有的变动都是轻而易举的：如改变分割卡的值，就要影响三个模块，CR (X1), INSPOOL (X 3) 和 JOB。

另外，还可对系统做很多有意义的改进，但会使系统结构有较大的变动。这些改进包括：

1. 用户作业也有权存取文件系统和多重虚拟输入和输出设备。
2. 在用户提交作业的过程中，用户文件应该始终保留不被删除。
3. 一种迅速从故障中恢复的检查点方法。
4. 建立一个已输入但尚未执行的作业的登录表，选择一种决定正在等待的作业的运行次序的调度方法。下一节中将进一步讨论这一问题。

在做这些改进工作时，遇到的问题之一是，在不能使用多重标记的情况下，就无法实现从属进程及其主进程之间的资源共享。看来需要重新定义从属进程，不必规定从属进程的字母表必须是主进程字母表的子集。这个题目留待进一步研究。

6.6 调度(Scheduling)

在大量用户共享有限的资源时，例如，用户数远大于资源数码，那么释放一个资源以前，往往已有着许多热切的用户在焦急等待。若在释放时，已有多个用户等候着，则应如何选择获取的用户是一个问题；以上各例中，采用的是非确定性的选择。对资源本身而言，如何选择是无关紧要的。但若在释放前，正好有一个进程加入了等待的进程集，而选择又是非确定性的，这个新加入的进程也许会被幸运地选上，设想这个资源十分抢手，而上述情形又一再发生。那末，就会有些进程一再被推迟，甚至

永远不能获取资源，或者至少不能预测要等待多久，这也是不能接受的，这种问题称为无穷抢占(见 2.5.5 节)[261]。

解决这个问题的一种办法是，保证所有资源不会过于频繁地被使用。为此，可以通过增加资源的数量，或合理使用资源，或提高使用资源的收费标准。这可能是处理任何时候都频繁使用的资源的唯一的解决方法。有些资源即使平均而言使用频率不高，但在高峰时期也会极其频繁。

有时可以试用不同时间不同收费标准来平缓用户要求。但这不一定总是成功的或者不太现实。在高峰期，平均而言，用户进程不可避免地要被延缓。对一个用户而言，更重要的是，操作系统如何保证这种延缓无很大波动，而且是可预测的：人们宁愿被告知在一小时内能得到服务，也不愿一直无法确知，自己究竟需等待一分钟抑或是一天。

在等待的用户中分配资源的任务叫作调度。为成功地进行调度，需要知道哪些进程进入了等待状态。为此，不应把资源的获取再看作是单一的原子事件，而需要分解为两个事件

please	请求分配资源
thankyou	实际分配资源进程

从 please 至 thankyou 所花的时间，就是它等待资源的时间。为区分要求分配资源的进程，我们将 please 至 thankyou 和 release 的每个出现赋予不同的自然数作为索引。进程获取它索引的办法等同于远程从属进程(6.4 节 X3)。如

$\square_i \geq 0$(res. i. Please;

res. i. Thankyou;

...;

res.i.release → SKIP)

调度资源的一个简单和有效的方法是，将资源分配给等得最久的进程。这种方法叫作先到先服务(FCFS)或先进先出(FIFO)。这同汽车站上排队的乘客所遵守的规定一样。

在面包房中，顾客不可能也不愿意排成一队，这就需要另一种算法以达到同样的效果。装置一部发票机，分发印有严格递增序号的票号。一进面包房，就取一张票，服务员一有空，就呼叫已取到票证但尚未服务的，序号又最小(早)的顾客。这个方

[261] 在操作系统领域，这个现象称为"starving"，饿死现象。

法叫作面包房算法。我们可以形式地表述如下。在描述中假设至多可同时为 R 个顾客服务。

例子

X1 面包房算法

我们需要三个计数变量

p　　已说 please 的顾客

t　　已说 thankyou 的顾客

r　　已释放他们的资源的顾客

显然，任意时刻，都满足 $r \le t \le p$。而且，p 随时会因新进入面包房的顾客而增加，r 则是随着新被服务的顾客而增大；p-t 是等待的顾客数，而 R+r-t 则是等待的服务员数。所有计数变量的初值为零，当全都相等时，如在夜间，最后一个顾客也已离开时，计数变量又都置为零。

该算法的主要任务是确保不会出现同时有自由的资源和等待的顾客；一旦出现这种现象，紧接的事件必须是 thankyou，即有个顾客得到了资源。

$BAKERY = B_{0,0,0}$

$B_{p,t,r}$ = if $0 < r = t = p$ then

　　BAKERY

else if $R + r - t > 0 \wedge p - t > 0$ then

　　$t.thankyou \rightarrow b_{p,t+1,r}$

　　else

　　$(p.please \rightarrow b_{p+1,t,r}$

　　$\mid (\square_i < t\ i.release \rightarrow B_{p,t,r+1}))$

面包房算法源自 Leslie Lamport[262].

[262] https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

第七章　讨论(DISCUSSION)

7.1 引言(Introduction)

我关于通信进程的研究的主要目标是试图找到满足下列特性的一个最简单的，可能的数学代数理论。

1. 这个理论可以刻画许多有趣的计算机应用，从自动售货机，过程控制，离散事件模拟，到资源共享的操作系统。

2. 这个理论可以在各种传统的或新颖的体系结构上高效的实现，例如，从基于微处理器的分时计算机到多处理器的互联网络[263]。

3. 这个理论可以清晰明了的协助程序设计员，完成对复杂的计算机系统的规约描述，设计，实现，验证和确认工作。

上面所有的目标都完美地达到基本上是不可能的。我一直希望，有可能有一种完全不同的方法，或者是某些细节部分的重大的改进，会导致上述一个或多个目标方面的更大的成功。本章中会讨论我和其他人曾经探索过的一些别的途径，而且我会解释一下我没有在 CSP 数学理论框架中采纳的理由。同时，我也借此指出和感谢这个领域中的其它研究工作者的一些开创性的，有影响力的工作。最后，我期待和鼓励人们在这个课题的基础理论方面以及其更广泛的实际应用方面，做出更深入的研究。

7.2 共享存储(Shared storage)

计算机并发式程序设计开始于 60 年代，是由当时的计算机体系结构和操作系统的发展所引出来的新问题。那时，计算处理能力非常稀少和珍贵。如果让处理器等待和慢速的外部设备通信，甚至和速度更慢的人类通信，那简直是莫大的浪费。因此，便宜的专用处理器（通道）就被用来独立地执行输入输出，从而使中央处理器有可能执行其它任务。为了使得昂贵的中央处理器总是在忙碌工作，采用分时操作系统，在计算机的主存中同时贮存若干个完整的程序，使得任意时刻总有一个程序在使用中央处理器，与此同时有若干个程序在使用输入输出处理器。在一个输入输出操作终止

[263] Hoare 的 CSP 著作成书于 1985 年，所以这里提的多处理器互联网络应该更多的是值 MPP 结构，而非现在的大规模分布式计算环境。

时，就会产生一个中断，使得操作系统能够考虑重新调度的决定，例如中央处理器下一步应该执行哪个程序[264]。

上述的描述说明，中央处理器和所有通道都应和计算机的主存相连；各个处理器对主存的存取操作是可能穿插进行的。而且，不同的程序是由不同的用户提交的不同作业，彼此间应是相互独立的。

因为这个原因，在设计硬件和软件时，我们设法将主存分成若干个不相交的片段，一个片段只由一个程序使用，而且确保程序间不会干扰各自的主存。为增加作业的吞吐量，也设法使同一计算机中有若干个独立的中央处理器[265]；若原来的操作系统具有良好的结构，则只要稍稍修改一下[266]，就能用于多处理器了，而对用户作业程序几乎不必修改。几个不同的作业共享单一计算机的缺点是

1. 使用的存储量与作业数成比例增加。
2. 除非最高优先级的作业，否则用户等待其作业运行结果的时间也要随作业数而增加。

可见，将单个程序分解为多个使用同一存储区的并发进程的组合，有可能使用户作业更好的利用硬件的并行性。

7.2.1 多线程(Multhreading)

第一个关于多线程的提议是基于转移命令(go to)。如果 L 表示程序中某处的标号，命令 fork L 将控制转到标号 L 处，但同时允许控制继续顺序往下传递。这样，就有两个处理器同时执行同一程序；每个处理器管理各自的控制轨迹，执行一系列命令。每条控制轨迹还可以分叉，故这种程序技术叫作多线程技术。

有了将程序一分为二的方法，随之也需要将两个进程合并的方法。非常简单的建议是用命令 join，只有当两个进程同时要求执行这个命令时，它才能被执行。先到达这个命令的进程，必须等待另一个进程也到达该处。然后，这个合并后的一个单一进程继续往下执行。

---

[264] 外设的 I/O 中断。另外一个常见的中断是时钟 Timer 中断。
[265] 例如，经典的 UMA 的共享内存的多处理机结构。
[266] 主要是需要注意一些 Kernel 中的数据结构的锁问题。

总的来说，多线程是及其复杂和容易出错的，除了最小的程序外，以不用为好[267]。可以存在一点辨解的是，这种技术是在使用结构化程序设计以前就提出来的；在那时，FORTRAN 还被看作是高级程序设计语言呢！

fork 命令的一种变种现在还在 UNIX™ 操作系统中使用。fork 不需要标号。它的作用是将分配给程序的整个存储区重新拷贝一份，然后将这份拷贝分给一个新的进程[268]。原先的进程和这个新进程一起自 fork 命令处往下执行。系统提供一种机制，使每个进程可以判断谁是它的双亲，谁是它的子孙[269]。给进程分配不相交的存储区，解除了多线程的主要困难和危险，但在时间和空间两个方面可能都不合算。这种方法适用于在作业的最外层引发（全局的）并发性，而不适用于小范围的并发性。

7.2.2 cobegin . . coend

为解决多线程中的问题，E. W. Dijkstra 提议：必须确保分叉后，两个处理器执行的程序块完全不同，不会彼此转移。令 P, Q 是这样的程序块，复合命令

cobegin P; Q coend

能导致 P 和 Q 同时启动，而且同时并行的执行，直到两者都结束后才终止。终止后，由单个处理器继续执行后随的命令。这种结构化的命令可用非结构化的 fork 和 join 命令来实现，我们可以使用标号 L 和 J，得到

fork L; P; go to J; L:Q; J: join

这种结构化的符号约定的一大优点是，容易搞清楚使用它的效果；特别当各块使用不同变量（这可用高级语言的编译程序检查或强制）时，更容易清楚。这时，称这些进程是不相交的；如果进程之间没有通信发生，则 P 和 Q 的并发执行的效果，和它们的顺序执行（次序任意）的效果完全一样。即

begin P; Q end = begin Q; P end = cobeginP; Q coend

而并行组合的正确性的证明方法，比之顺序情形甚至更为简单。这也是为什么本书的并行构造基于 Dijkstra 的建议。与其不同的只是记号；为避免与顺序组合相混淆，我用算子‖将进程分开；这样就可以使用一般的括号把进程符号等包括起来，而不需要使用更麻烦的 cobegin . . . coend 。

---

[267] 多线程的编程模型现在已经很普及了。这与操作系统领域 POSIX 的发展和标准的制定很有关系。

[268] fork 产生了一个新的 Unix Process，而非共享内存的线程。

[269] 通过 fork()的返回值来判断。如果成功 fork 一个子进程，fork 命令返回给父进程子进程的 PID；返回给子进程 0。

7.2.3 条件临界区(Conditional critical region)

限定并发进程间不能共享变量，使得进程间无法通信或者交互作用，这个限制严重地减低了并发性的潜在价值。

本书中引入的（模拟的）输入和输出通道是解决上述问题的一个方法；但早期解决这个问题的技术是由计算机的硬件联想到的，即用共享主存的办法实现并发进程间的通信。Dijkstra 曾说明可用由互斥信号量保护的临界区（6.3 节）来达到这一目标。后来，我也曾建议在高级程序设计语言中将其形式化表示。在若干个共享进程的临界区中更新的一组变量，应被申明为一种共享的资源，例如

```
shared n : integer
shared position : record x, y : real end
```

更新这种变量的临界区的前面应写上 with 子句，子句中引用这些变量的名字，如

```
with n do n： =n + 1;
with position do begin x := x +deltax； y := y + deltay end
```

这种符号约定的好处是，编译程序能自动地引入必要的信号量，围绕每个临界区能填上必要的 P 和 V 操作。而且在编译时刻可以检查，是否对共享变量的存取和更新只出现在由相关信号量保护的临界区中。

共享存储的进程的合作方式是多样的，可能要求其它同步形式。如，某一进程更新变量的目的是要其它进程读取这个新值，故在更新变量前，其它进程就不应该读取。反之，在其它进程尚未全部读取更新值前，不应再更新变量。

解决这个问题，可用条件临界区。其形式为

```
with sharedvar when condition do critical region
```

在临界区的入口，先检验条件。若条件为真，就执行临界区；若假，就推迟临界区的进入，从而其它进程允许进入它们的临界区，并更新共享变量。每做一次更新后，重新检验条件。若条件变为真，则被推迟的进程进入其临界区；否则该进程重新挂起。若多个推迟的进程都能前进，它们间的选择则是任意的。

若有多个进程更新并读取一个消息，则需申明一个整数变量作为资源的一部分，并由它统计在更新读取消息的进程的个数，如

```
shared message : record count : integer; content : . . . end ;
message.count := 0;
```

包含临界区的更新进程则为

```
with message  when count = 0 do
        begin content := . . . ;
                . . .;
        count ： =number of readers
        end
```

每个包含一个临界区的读取进程则为

```
with message when count > 0 do
        begin my  copy := content; count = count -1 end
```

条件临界区也可用信号量实现。和由程序员直接使用同步信号量相比，条件临界区的开销是非常大的；每次退出一个临界区，必须重新测试进入临界区等待进入临界区的全部进程的条件。幸运的是，条件不需要频繁地进行重新测试，因为限制访问共享变量可以确保等待进程测试的条件只有在共享变量本身更改值时才能更改值。条件中的所有其他变量对于等待过程必须是私有的，在等待过程中显然不能改变。

7.2.4 管程(Monitors)

管程的提出来自 SIMULA 67 的 class，它本身又是 ALGOL 60 的过程概念的推广。基本想法是，关于数据的全部有意义的操作（包括初始化及终止化），应该与申明数据本身的结构和数据类型收集在一起；当共享这个数据的进程需要使用某操作时，就用过程调用的办法来启用它。管程的一个重要特征是，每个时刻每个管程中只有一个过程体处于活动状态；若两个进程同时调用一个过程（或者调用两个不同过程），其中的一个调用必须推迟到另一个调用完成为止。过程体就象是由相同的信号量保护的临界区。

例如，计数变量的一个简单管程，用 PASCAL PLUS 的记法，其形式为

```
1  monitor count;
2  var n : integer;
3  procedure* up; begin n :=n + 1 end ;
4  procedure* down; when n > 0 do begin n := n-1 end ;
5  function* grounded : Boolean; begin grounded := (n =0) end;
6   begin n := 0;
7   ...;
```

```
8       if n ≠ 0 then print(n)
9  end
```

上述程序的一个解释：

行 1　申明管程及其名称 count。

行 2　申明管程里的局部的共享变量 n。只有在管程中才能存取它。

行 3-5 申明管程中的三个过程。只有使用该管程的程序才能存取它。

行 6　管程由此开始执行。

行 7　三个粗点是一些内部语句，表示使用管程的程序块。

行 8　从用户程序块退出时，将 n 的终止值（若非零时）打印出来。

可以申明管程的新的实例，使其局部于程序块 P，如

```
        instance rocket : count; P
```

在程序块 P 中，可用下列命令调用带星号的过程

```
        rocket.up; ... rocket.down; ...;  if rocket.grounded then ...
```

在 P 中不能使用不带星号的过程，也不能存取不带星号的变量，例如 n。这些规定可由编译程序强制执行。管程以固有的互斥性，保证管程中的过程可由 P 中的任意多个进程所调用，而且确保在更新 n 时不会产生干扰(不一致性)的危险。注意，若在 n=0 时调用 rocket.down，这个调用就会被推迟(从操作系统原语的角度，调用进程会进入一个睡眠队列)，直到 P 中的其它进程调用 rocket.up 后才会重新执行。这样，确保 n 不会取负值。

申明一个管程实例的效果可由 ALGOL60 中过程调用的复制规则来解释。先复制一份管程的文本；然后将程序块 P 复制到管程中的第七行的三个点处，并将管程中的全部局部名冠以实例名，得到

```
rocket.n : integer;
procedure rocket.up; begin rocket.n := rocket.n + 1 end ;
procedure rocket. down;
        when rocket.n > 0 do begin rocket.n := rocket.n — 1 end ;
function rocket.grounded : Boolean;
        begin rocket.grounded =(rocket.n =0) end ;
 begin rocket.n ： =0;
        ...；if rocket.n ≠ 0 then print(rocket.n)
 end
```

复制规则使用户进程不会忘记将 n 的值初始化，也不会忘记打印其终止值，或者在合适的时候打印。

反复检验入口条件会造成低效率，故在管程的设计中引进了更精巧的机制，可以让进程等待，或者让等待的进程继续运行。使用这些机制，甚至可在过程调用的执行中途挂起自身，在自动解除排它性后，另一进程的过程调用又可将挂起的进程唤醒，使其继续执行。用这种方法可以高效地实现很多漂亮的调度技术；但我现在不认为引入这些复杂机制是值得的。

7.2.5 管程嵌套(Nested Monitors)

一个管程实例亦可用作信号量来保护单个资源，如打印机；每个时刻这种资源只能被一个用户所使用。这类管程可写作

```
monitor singleresource;
var free : Boolean;
procedure * acquire;
when free do free := false;
procedure * release; begin free true end ;
begin free true; ... end
```

但是，由上述管程所提供的保护可能被破坏，例如，进程可以不经获取，就直接使用资源；也可以使用后忘了释放它。为避免这种危险，可使用类似于 6.4 节 X4 的虚拟资源的一种程序结构。这种结构亦形如管程，但它局部于实际资源管理。虚拟资源的名称上冠以星号，表示可由用户进程存取。而 * acquire 和* release 上的星号则删去，使其只能在虚拟资源管程中使用，而不能由其它进程误用。详情如下

```
monitor singleresource;
        free : Boolean;
        procedure acquire;
                when free do free := false;
        procedure release;
                begin free := true;
                end
        monitor * virtual;
                procedure * use(l : line); begin .. . end ;
                begin acquire; ... ; release
```

```
        end
        begin free :=true; …
        end
```

这个管程的一个实例可以声明如下：

```
instance lpsystem : singleresource; P
```

在程序块 P 中，向行式打印机输出一个文件，写为

```
instance mine : lpsystem.virtual;
    begin . . . mine.use(l1); . . . mine.use(l2); . ..end
```

对使用打印机必须的获取和释放操作都由虚拟管程自动地插入于用户程序块的前后，这样就防止了不讲公德的用户对打印机的滥用。原则上，用户程序块也可分成若干个并行进程，它们共同使用虚拟管程的实例 mine，但在这里不讨论这种使用方式。在 PASCAL PLUS 中，称只被一个进程使用的管程为闭体(envelope)；这种管程可以更高效地实现，不必涉及排它性和同步；而且可由编译程序检查其是否无意地被共享了。这些实例申明的含义同于复制规则的重复应用，表示如下

```
var lpsystem.free : Boolean;
procedure lpsystem.acquire;
    when lpsystem.free do lpsystem.free := false;
procedure lpsystem.release;
begin ipsystem.free := true end ;
begin ipsystem.free := true
    .
    .
    .
    begin
    procedure mine.lpsystem.use(l : line); begin ... end;
        lpsystem.acquire;
        ... mine. lpsystem. use( 11 ) ;
        ... mine. lpsystem. use( 12) ;
        lpsystem.release;
    end
    .
    .
    .
end
```

上面列出的拷贝的细节，是为了给初学者解释虚拟管程的含义；而有经验的程序员根本不愿读这个繁琐的程序，甚至不必去思考这个拷贝过程。

上述符号约定是在 1975 年，当时用来解释一个类似于 6.5 节中的操作系统；后来又用 PASCAL PLUS 实现了这些管理。混合使用星号和嵌套，可以取得迹极巧妙的效果；但是基于 PASCAL 和 SIMULA 的这套符号是非常笨拙的；用替代及重新命名来解释它们的含义，也是令人费解的。 Edsger W. Dijkstra 当年对它们的批评，促使我去设计通信的顺序进程。

但是，由 6.5 节中的结构可清楚看出，控制共享的机制本来就是复杂的；无论在通信进程的框架中表示，或者在 PASCAL PLUS 的拷贝规则和进程调用的语义中表示，都是复杂的。选用哪种语言，取决于人们的口味，或者是效率的考虑。在共享主存的计算机上实现一个操作系统，也许 PASCAL PLUS 具有更多的长处。

7.2.6 Ada™

Ada 中用于并发式程序设计的成分是 PASCAL PLUS 的远程过程调用和非结构式的输入输出通信的混合物。进程称作任务，它们间的通信是由 call 调用语句和 accept 接受语句实现；前者如同带有输出输入参数的过程调用语句，后者从语法和效果上都类似于过程申明。典型的接受语句为

```
accept put(V : in integer; PREV : out integer) do
PREV = K; K := V end
```

相应的调用语句可以为

```
put(37, X)
```

标识符 put 称为入口名。

设在不同任务中有两个名字相同的调用语句和接受语句，当两个任务都准备好执行它们时，它们才能被执行。执行的效果如下

1. 将调用语句中的输入参数复制到接受任务
2. 执行接受语句的程序体。
3. 将输出参数的值送回调用语句。
4. 然后，从它们的下条语句开始，继续执行各任务。

执行接受语句中的程序体的过程叫作汇合(endezvous)，就象调用任务和接受任务在一起执行它。汇合机制是 Ada 的一个非常引入注目的特色，汇合机制简化了经常出现的交互式输出输入现象时的处理方式，而在只需要单独的输入或输出时情况也不会复杂。Ada 中的挑选语句类似于 CSP 的选择算子□，其形式为

```
select
        accept get(v : out integer) do v := B[i] end ; i := i+1; ...
or accept put(v : in integer) do B[j] := v end j := j+1; ...
 or ...
end select
```

运行时，由 or 隔开的分支恰好只有一个分支被选中执行，这种选择依赖于调用任务。在完成汇合后，和调用任务并发执行的，是选中分支中 accept 的 end 后的语句。我们亦可用 when 来规定分支的选择条件，如

when not full ⇒ accept...

其效果如同条件临界区。

挑选语句的分支不一定都以 accept 开始，也可用 delay。如果事先规定了延迟的时间的长短，若经过规定时间仍无分支被选中，则选择以延迟(delay)开始的分支。这样可以避免，由于硬件或软件的错误可能造成的死等的危险。在我们的数学模型中有意不提时间，故不可能确切地表示延迟现象[270]；除非规定分支的非确定性选择都带有一定的时间延迟。

挑选语句的分支亦可以为 terminate 语句[271]。当调用该任务的各个任务都已终止时，才选择这个分支；从而使该任务亦终止。这种办法不如 PASCAL PLUS 中的内部语句允许管程在终止时做些扫尾工作。

挑选语句中还可有子句 else，当不能立即选择其它分支时，就选用它；一个分支不能被立即选上的原因，可以是它的 when 条件为假。或者在其它任务中没有相应的调用语句已在等待执行。else 似乎等价于时间为零的延迟。

调用语句亦可使用 delay 或 else 来保护自己，免得延迟过久。但这样做的结果可能会降低分布式计算机网络的效率。

---

[270] CSP 理论不考虑时序逻辑系统中的 Liveness 和 Safeness 这些属性。CSP 更偏向于一个支持并发成分的 Programming Language 的理论。

[271] 有点类似 Garbage Collection 的程序块。

Ada 中申明任务的方式很象 4.5 节中附属进程的申明方式；但每个任务可服务于任意多个调用进程，这一点又同于 PASCALPLUS 中的管程。另外，程序员还必须保证任务能终止。任务的定义分成两部分，描述部分和任务体。描述部分包括任务名，以及全部调用入口名和参数类型。使用任务的程序员及编译程序都必须了解这部分信息。任务体规定了任务的行为，可以和使用该任务的程序分别编译。

每一个 Ada 的任务都需要指定一个固定的优先级。若可前进的任务数多于有效的处理器数，则低优先级的任务就被搁置。执行汇合的优先级高于调用任务和接受任务。优先级的表示叫作 pragma；可用于加速关键任务的响应时间，但不影响程序的逻辑行为。将程序的抽象的逻辑正确性和其实时响应问题分开处理，是一种极好的想法；用实验或选择适宜的硬件解决实时响应问题，经常是比较容易办到的。

Ada 中还提供别的辅助功能。例如可检测有多少个调用任务正等候进入某个入口。一个任务可以突然终止(abort)另一个任务，而所有任务都仰赖于这种任务。任务可存取和更新共享变量。编译程序可延迟，重新排序，或者合并变量的更新，根本不考虑变量是否是共享[272]；这样的油画效果是无法预测的。还有一些其它功能，带来更多的复杂现象和交互影响，这里我不再介绍了。

除了上面提到的那些复杂问题外，Ada 的任务化功能很适合用于共享存储的多机系统的实现和应用的。

7.3 通信(Communication)

由于计算机硬件的惊人的进展，刺激人们去探索构一个通信的进程网络。微处理机的发明将处理能力的价格降低了若干个数量级。而和微处理机相比传统的计算机仍然相当昂贵。看来，用几个微处理机合作解决一个任务是最经济的增大处理能力的方法。这些微处理机之间用电线相连，它们沿着这些线路相互通信。由于每个微处理机有自己本地的主存，存取速度极快；这就避免了多个处理机共享主存，而每个时刻只能由一个处理机拜访这个共享主存所造成的瓶颈口问题[273]。

[272] 编译的各种优化。

[273] 这里提出的是一种 NUMA 结构。每个处理器都有自己的局部存储器。处理器之间通过互联网络连接。

7.3.1 管道(Pipe)

如 4.4 节所述，处理单元之间最简单的通信模式是单向消息在每个进程与它的邻居之间在线性管道中传递。头一个提出这个想法的人是 Conway，他解释这个想法所用的例子类似于 4.4 节 X2 和 X3，除了管道中的各个部件都必须能成功终止，不能永远运行。他提出可用管道结构编写程序设计语言的多遍编译程序。在有足够存储空间的计算机上，同时激活各遍编译过程，在各遍之间通过管道传递控制和消息，以此模拟并行执行。如果存储空间不足时，每次只能激活一遍编译，而将其输出的数据存放在备份存储器中的文件中。这一遍完成后，再启动下一遍，并从文件中输入前一遍产生的数据。尽管这两种的执行方式十分不同，但编译的结果完全一样。这说明了程序设计的一种抽象特征，即在程序设计中允许不同的实现方法，以适用于不同的环境，但达到相同的效果。Conway 的建议适用于存储大小不一的各种计算机上，对软件的实现是很有价值的。

管道 pipe 也是 UNIX™ 操作系统中的一个标准通信方法，但 Unix 中使用记号‘|’，而不是‘>>’。

7.3.2 多重缓冲通道(Multiple buffered channels)

管道结构只允许进程链的进程间单向通信，不关心消息序列是否需要缓冲。对管道的一种自然推广是进程间可以双向通信，为通信通道提供缓存看来也是自然的。在操作系统 RC4000 的内核中实现了缓冲通信机制；使用内核中的缓冲通信机制，为模块间提供通信，在一个高层次上提供服务。在更宏观的层面，例如美国 APPAnet 那类存储转发的报交换网络中，在消息源和目的地之间不可免地要插入缓冲装置。

当进程间的通信由线性链式通信推广为网络状通信时，就可能出现通信回路，这时有无缓冲装置会导致系统的逻辑行为的巨大区别。缓冲并不总是可取的：例如，可以写出一个程序，当通信的缓冲长度超过 5 时就会死锁；也能写出一个程序，除非缓存长度超过 5，否则一定死锁。为避免这种无规律现象，所有的缓冲长度都应该是无界的。可惜，这又会导致严重的低效实现问题，主存中将充斥被缓冲的消息。而且也会影响数学处理，即使每个进程部件只有有穷个状态，整体网络系统也由无穷缓冲机制而成为无穷状态机。对于人机间的高速和可控的交互作用而言，缓冲也是一种障碍；缓冲会在信号源和响应间插入延迟。一旦在加工缓存的信号源中发生错误，很难

跟踪和恢复。缓冲是一种批处理技术，在需要高速响应而不是大量计算的情况下，不宜使用。

7.3.3 函数式多道处理(Functional multiprocessing)

一个确定性进程可以通过一个数学函数来定义。该函数将输入通道映射到输出通道。通道看作是通道上传递的可不断增长的消息序列。这类函数可通过输入序列的递归结构来定义，但不考虑空输入序列。例如，将每个输入的数乘以 n，并输出结果的进程，可定义为

$prod_n(left) = <n \times left_0>^{\wedge} prod_n(left')$

以两个已排序的（无重复数据的）数据流作为输入参数，输出合并后的排序的（去掉重复数据的）单数据流的函数，可定义为

merge2(left1, left2)=

if $left1_0$ < $left2_0$ then

$<left1_0>^{\wedge}$merge2(left1' , left2)

else if $left2_0$ < $left1_0$ then

$<left2_0>^{\wedge}$ merge2(left1, left2')

else

$<left2_0>^{\wedge}$ merge2(left1' , left2')

一个无回路网络是这类函数的复合函数。例如，合并三个已排序的输入流的函数是

merge3(left1, left2, left3) = merge2(left1, merge2(left2, left3))

图 7.1 为这个函数的网络示意图。

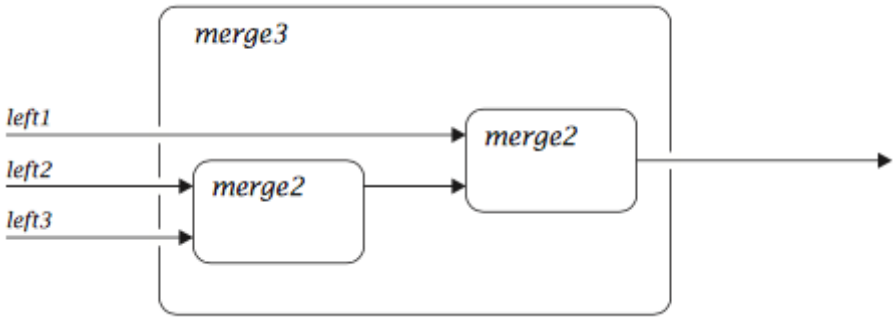


图 7.1

有回路网络则可由递归方程组定义。例如，Dijkstra 提供给 Hamming 的问题，按升序输出非平凡素数因子为 2，3 或 5 的全体数字。第一个这样的数字是 1；若 x 是这

类数字，2×x，3×x 和 5×x 也是这类数字。用进程 $prod_2$，$prod_3$, $prod_5$ 产生这些乘积，再由进程 merge3 排序，见图 7.2.

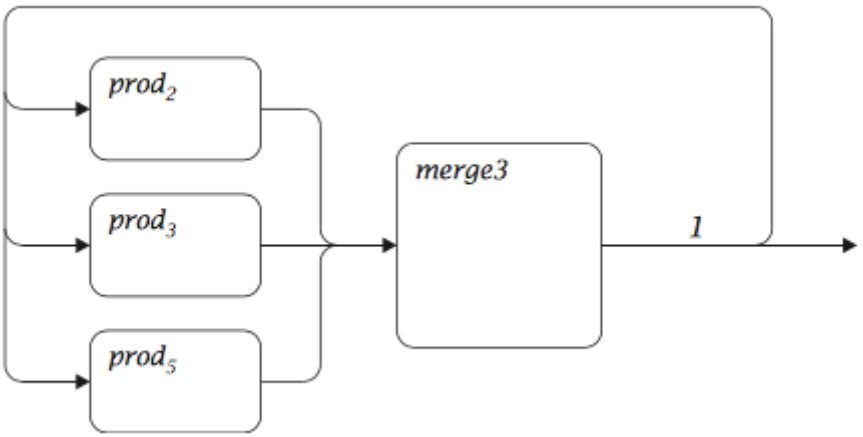


图 7.2

所求函数是无输入的；其定义为

Hamming =

<1>^merge3($prod_2$ (Hamming) , $prod_3$ (Hamming) , $prod_5$ (Hamming))

多处理机网络的函数式表示方法，在下列方面与本书中的方法有很大区别，即

1. 实现时，一般要求所有通道都有无界缓冲装置。
2. 输到缓存中的每个值要一直保持到所有输入进程都取到它后，才能删去。
3. 不允许进程等待两个输入之一，而且取任一个先到者。
4. 所有进程都是确定的。

最近的研究已部分解决了(1)和(2)引起的低效率问题，放松了(3)和(4)的限制。

7.3.4 无缓冲通信(Unbuffered communication)

多年来，我选用无缓冲(同步)通信作为基本的通信方式。其理由为

1. 这与用电线作为物理介质连接通信体是相配的。而电线是不能存储消息的。
2. 这与在单处理器中通过拷贝参数和结果，调用子程序并返回的效果是相当的。
3. 需要缓冲时，可用进程实现；而且缓冲的深度亦可由程序员严格控制。
4. 7.3.2 节结尾处提及的缓存的缺点，也是不选用的理由。

当然这些理由不能使人绝对信服。例如，若以缓冲通信作为原始通信方式，则不必严格要求子程序调用和返回必须交替出现了；同步亦可用其实现，只需在每个输出后跟之以输入一个确认，和每个输入后跟之输出一个确认。

7.3.5 通信顺序进程(Communicating sequential processes)

我第一次全面解释一种基于并发和通信的程序设计语言时，就用通信顺序进程(Communicating sequential processes)作为标题，那个早期的建议和本书相比较，有两个重要的不同。

(1) 并行组合

早期的通道没有名字。但组成并行结构的进程具有名字，不同的进程的名字不同，每个进程名字之前，用一对冒号隔开加一个前缀，例如:

[a::P ‖ b :: Q ‖. . . ‖ c :: R]

在进程 P 中，命令 b!v 将值 v 输向名字为 b 的进程。进程 Q 中则用命令 a?X 输入这个值。进程名字局部于它们所出现的并行组合，并行组合中进程间的通信是被藏匿的。

这种写法的优点是，在语言中不必引入通道或通道申明的概念。而且在逻辑上就不会破坏对通道的下述限制，即一个通道只单向连接两个进程，其一用其输入，另一用其输出。这样的限制但在实用和理论方面也有缺点。

1. 实用上的严重缺点是，从属进程需要知道使用它的进程的名字；这就使建立从属进程库的工作很复杂。
2. 数学上的缺点是，并行组合不再是一个可结合的二元算子，而是一个带有可变多个参数的运算了。

(2) 自动终止

早期版本中，并行命令中的所有进程都应终止。这个规定的理由是，希望能用传统程序设计中的后置条件来描述进程的正确性，也就是在进程成功终止时，某个谓词应该为真。（这个希望却从未得到满足，而其它证明方法，例如，1.10 节，现在看来是更为不错。）从属进程必须终止的规定造成的后果是，使用进程终止时必须通知其从属进程。这样就引起了一个特别的约定，即循环

* [a?x → P □ b?x → Q □ ...]

自动终止的条件是，向它输出消息的进程，如进程 a，b，...，都终止。这样，从属进程在终止前就可完成必要的终止化工作；这是 SIMULA 和 PASCAL PLUS 的已被实践证明的十分有用的特征。

这个约定增加了定义和实现的困难；程序正确性的证明方法也更为复杂。在 4.5 节中，我放宽了简单从属进程必须终止的限定；对更复杂的情形，采用了别的方法（见 6.4 节）。

7.3.6 Occam

不同于 Ada，occam 是一种非常简单的程序设计语言，非常严格遵循本书中提出的各个原则。最大的不同只是符号而已；Occam 借助于语法检查编辑程序，可在屏幕上组合 occam 的语法对象；它不使用插入式的算子符，而使用前缀式运算，它不用括号，而用缩排。

```
SEQ    表示(P; Q; R)
    P
    Q
    R
PAR    表示(P ‖ Q ‖ R)
    P
    Q
    R
ALT    表示(c?x → P □ d?y → Q)
    c?x
        P
    d?y
        Q
IF     表示(P⊀B⊁Q)
    B
        P
    NOT B
        Q
WHILE  B   表示(B*P)
        P
```

ALT 对应于 Ada 中的挑选语句，并提供类似的选项。可用布尔条件 B 限制某分支的选用，如

```
B＆c?X
P
```

上述的输入命令亦可用 SKIP 替代，这时该分支的选择完全由布尔条件所限制；若代之以等待，则该分支只能在度过规定的某段时间后才能被选用。

Occam 中没有关于导管(4.4 节)，从属进程(4.5 节)，和共享的进程(6.4 节)的专用记号。各种通信方式都用通道名字相同的输入输出通道上的通信来实现。为此，过程申明中可以通道为参量，例如，简单的复制进程可申明为

```
PROCcopy（CHANleft，right）=
    WHILE  TRUE
        VARx：
        SEQ
            left?x
            right!x：
```

而两单位的缓存 GOPY >> COPY 则由如下构成

```
CHAN mid：
PAR
    copy（left, mid）
    copy（mid, right）
```

N 个缓存的链，则可用一个 n 个通道的数组和并行结构的一种迭代方式所构成；这种并行迭代通过 n-2 个进程所构成。并行组合通过下标为 i 的 n-3 个进程所组成。

```
CHANmid[ n-1]：
PAR
copy(left，mid[0])
PAR_i=[0 FOR n-2 ]
copy(mid[ i], mid[i+1])
copy(mid[ n-1], right)
```

由于 occam 是在个数确定的处理器上，用静态存储分配技术实现的，故上例中的值 n 必须是一个常数。由于同样的原因，也不能用递归过程。

在 occam 中，为得到从属进程，也需要用类似的构造，如

```
PROCdouble（left，right）=
    WHILE TRUE
        VARx：
        SEQ
            Left?x
            right!(x+x)：
```

这个程序可由单个用户进程 P 申明为其从属进程，如

```
CHAN doub.left, doub.right:
PAR
        double（doub.left, doub.right）
        P
```

在 P 中，将数加倍的方法是

```
doub.left!4; doub.right?y; …
```

用通道数组及 ALT 的迭代形式可实现共享的进程，每个用户进程使用通道数组的一个元素。例如，设计一个积分器，每输入一个新的数据，就输出迄今输入的所有数据的和数。令所要的进程为

```
CHAN add，integral[n-1]:
PAR
        VARsum，x:
        SEQ
                sum :=0
                ALT_i = [0 FOR_n]
                        add[i]?X
                        SEQ
                                sum := sum+x
                                Integral[i]!sum
        PARi = [0 FOR_n]
                ...user processes…
```

类似于 Ada，occam 中亦可对并行组合中的进程设置优先级。只要将 PAR 换成 PRI PAR，则位置在前的进程有较高优先级。屏幕编辑程序还提供重排进程优先级的编辑命令。对 ALT 构造，也提供类似的优先级安排，即有 PRL ALT。若有多个分支可立即选用，则写在最前面的分支被选用；其它方面的效用如 ALT 语句。当然，程序员必须保证他的程序的逻辑正确性是和优先级的设置无关的。

occam 还另有在处理器间分配进程的手段，以及规定处理器的物理插针和程序中通道的对应的手段，包括规定输入程序的插针。

7.4 数学模型(Mathematical models)

在 60 年代初，人们开始认识到程序设计语言应该有精确的数学语义。数学语义给予语言一种安全的、无歧义的、精确的、稳定的描述；这种描述可用作语言的用户和实现者之间双方都同意的一种接口。也是不同的实现是否实现了同一个语言的评判的必要的依据。数学语义是计算机语言标准化的关键，就如同螺栓螺母标准化中所用到的那些计量。

60 年代后期，人们发现了数学语义更重要的作用，它有助于程序员履行保证其程序正确性的义务。R.W.Floyd 提出，语义是一组正确的证明规则，而不是一个显示的，清晰的数学模型。PASCAL，Euclid 和 Gypsy 的规约中就采用了这一建议。

通信顺序进程早期的设计（见 7.3.5 节）中没给出数学语义，结果很多重要的设计问题无法解决，例如

1. 是否可在并行命令中嵌套另一并行命令？
2. 如果可以，则能否用递归过程并行地调用自己？
3. 理论上是否可以卫式的输出命令？

本书中的数学模型给上述问题以肯定的回答。

7.4.1 通信系统演算 CCS

并发性的数学模型的主要突破是由 Robin Milner[274]完成的。他研究的目标是给出一个构造和比较不同描象程度的不同的模型的框架[275]。他通过表达式的基本语法为起点用于表示进程，然后他定义了表达式间的一系列等价关系，其中最重要的有

强等价(Strong equivalence)

观察等价(Observational equivalence)

观察同余(Observational congruence)

每一种等价关系定义并发性的一个不同模型。CCS 通常以观察同余得到的模型作为进程间相等的最基本的定义。

---

[274] Robin Milner(13 January 1934 – 20 March 2010)，著名计算机理论科学家，1991 年图灵奖获得者，英国爱丁堡大学理论计算机实验室 LFCS 的创办人之一。主要学术贡献为第一个自动定理证明语言 LCF 和并发系统理论 CCS，pi-calculus 等。

[275] 关于 CSP，CCS 的符号系统和一些基本算子的比较，可参阅 Colin Fidge 写的"A Comparative Introduction to CSP, CCS and LOTOS", 1994。

CCS 的基本记号解释如下

a.P	对应于 a → P
(a.P) + (b.Q)	对应于(a → P ∣ b → Q)
NIL	对应于 STOP

CCS 与 CSP 在符号系统上的不同更为重要的差别是在对隐匿(hiding)的处理上。在 CCS 中，用特殊符号 τ 表示一个隐匿的事件或者一个内部的转移。使用这个记号记录隐匿事件的记录的好处是，可用其构造卫式递归方程(Guarded Recursive Equations)，从而保证解的唯一性，例如，2.8.3 节的说明。另一个（但略微不那么重要）好处是能执行 τ 的无界序列的进程不一定都能归到 CHAOS；这样就可区分发散进程。但是 CCS 不能分辨可能发散的进程和行为类似的但不是发散进程的区别。因此，我觉得，这个问题可能导致一个完整的 CCS 语言实现是不太可能的。

CCS 不提供非确定性⊓算子。但可用 τ 符号来模拟非确定性，如

(τ.P) + (τ.Q)	对应于 P ⊓ Q
(τ.P) + (a.Q)	对应于 P ⊓ (P ⊓ (a → Q))

但这些对应并不确切，因为由 τ 所定义的 CCS 的非确定性不满足 CSP 中定义的结合律[276](associative)，例如，图 7.3 中的树是各不相同的。

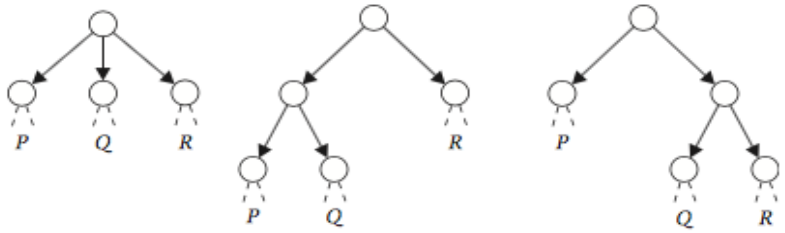


图 7.3

另外，前缀对非确定算子也不满足分配律[277]，如图 7.4 中的树亦是不同的（设 P≠Q）。

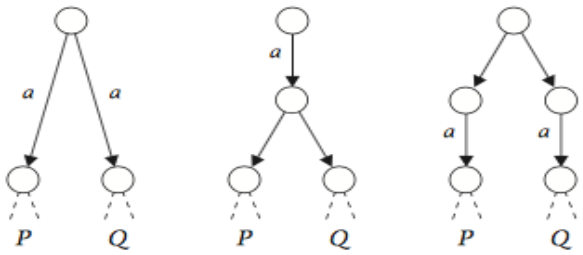


图 7.4

[276] 读者可参阅 3.2.1 的 L3。P ⊓ (Q ⊓ R) = (P ⊓ Q) ⊓ R。

[277] 读者可参阅 3.2.1 的 L4。x → (P ⊓ Q) = (x → P) ⊓ (x → Q)。

这些例子表明，CCS 尽量区分各种进程；在 CSP 书中视为行为相同的进程，在 CCS 中是不同的。其原因在于，CCS 的目标是建立许多模型的一个最基本的框架，每种模型的引入可扩大 CCS 中进程的等同性，而不是减少。为避免限制模型的范围，故 CCS 只采用最基本的等同性作为基础。本书的数学模型理论追求的恰好是相反的目标—只保持最基本的区别,而定义尽可能多的等同性。因此 CSP 可以给出更为丰富的代数法则。我们希望这些法则在论证设计和实现时能发挥实际作用；能允许使用比 CCS 更多的程序变换和优化技术。

CCS 基本的并发组合算子记作单竖|。它比 CSP 的组合算子‖更复杂。它包含了隐匿(hiding)，非确定性(nondeterminism)，交叉(interleaving)及同步(synchronisation)。

CCS 的每个事件有两个形式,简单形式(a)或带有杠的(ā)。两个进程在一起并发运行时，仅当一个进程要执行带杠事件，而另一个要执行相应的简单事件，才出现同步现象。它们合作完成的事件立即被隐匿起来，代之以记号 τ。然而，同步不是强制的；两个事件又都可以各自独立地和明显地出现，作为和外部环境的一种交互作用。在 CCS 中，一些定义如下：

(a.P)|(b.Q) = a.(P | (b.Q)) + b.((a,P)|Q)

(a.P)|(a.Q) = a.(P|(a.Q)) + a.(a.P)|Q)

(a.P)|(ā.Q) = τ.(P|Q) + a(P|(ā.Q) + ā.((a.P)|Q)

因此，CCS 只允许两个进程执行一个同步事件；若两个以上进程都准备就绪,哪对进程能成功地同步执行，则是非确定性的。如

(a.P)|(a.Q)|(ā.R) = τ.(P|(a.Q)|R) +

τ.((a.P)|Q|R) +

a.(P|(a.Q)|(ā.R)) +

a.((a.P)|Q|(ā.R)) +

ā.((a.P)|(a.Q)|R)

由于 CCS 增加了并行算子的复杂性，故无需屏蔽算子。但 CCS 引入了有限制算子\，其作用为阻止有关事件的出现，并从字母表中将所有有关事件及共带杠变种删去。下列 CCS 的法则可解释其效用

(a.P) \ {a}= (ā.P) \ {a} = NIL

(P + Q) \ {a} = (P \ {a}) + (Q \ {a})

((a.P) | (ā.Q)) \ {a} = τ.((P|Q)\{a})

((a.P) | (a.Q) | (ā.R)) \ {a} = τ.(P | (a.Q) | R) \ {a}) +

$$\tau.(((a.P) \mid Q \mid R) \setminus \{a\})$$

上述最后一个法则解释了如何使用 CCS 的并行组合算子达到共享进程的效果，即(a.P)和(a.Q)共享(ā.R)。CCS 的目标之一是，用尽可能少的基本算子达到最强的表达能力。这也是 CCS 优美且有力的原因之一，大大简化了通过等价关系所定义的各种模型的研究。

在本书中，我采用的是一种与 CCS 互补的方法。用单一的简单模型来达到简单性，使用这个模型很容易地定义很多算子，只要有益于刻画不同概念的算子我们都引人。例如,非确定的选择算子⊓用来介绍最纯粹的非确定性，完全独立于环境选择$(x{:}b \rightarrow P(x))$。类似地，‖介绍并发和同步，完全独立于非确定性及藏匿，各有各的算子。由算子所对应的概念的清晰性，达到了算子所遵循的代数法则的简明性。为使数学理论实用，需要引入足够多的算子。当然，寻求极小的算子集在理论研究方面也是有用的。

Milner 还引人模态逻辑(Modal Logic)描逃进程的可观察的行为。模态词

$$\langle a \rangle S$$

刻画了一个进程，它可能完成 a，然后按 S 描述的行为动作。而其对偶

$$[a]S$$

也刻画了一个进程,它若以 a 开始，则以后必须如 S 动作。CCS 建立了证明正确性的演算；进程 P 满足描述 S，用传统的逻辑记号表示为

$$P \models S$$

这个演算完全不同于关于 sat 关系的演算，它基于规约的结构，而不是程序的结构。如否定词的规则是

$$\text{若 } P \models F \text{ 不成立 则 } P \models \neg F$$

这意味着，在开始证明进程 P 的正确性时,必须先写下这整个进程。反之，使用 sat 时，就可由分进程的正确性的证明构造复合进程的正确性证明。而且从不需要证明一个进程不满足其描述。模态逻辑在理论上是十分有趣的,但对通信进程而营，迄今还没看到有意义的应用。

CCS 中的等同性是一种强等同关系，两个相等的进程不仅有相同的可观察的行为，而且有相同的被隐匿的行为。因此,CCS 是表达和研究各种弱等价定义的良好模型，这些弱等价关系会忽略更多的隐匿部分。 Milner 为此引人了观察等价的概念。这涉及如何定义对进程进行观察和试验；两个进程中任一个的可观察的行为，也是另一个进程的可观察行为，则这两个进程是等价的；这其实是哲学上的不可分辨物的等同

性原理的一个应用。这个原理也是本书的数学理论的基础，这个数学理论将进程等同于对其行为的观察的集合[278]。等同性原理可象征地表示为，两个进程 P 和 Q 等价的充要条件是它们满足相同的形式描述,即

$$\forall S \bullet P \models S \equiv Q \models S$$

可惜，这个简单的等价定义不尽满意。如果两个进程被认为等同，用同一个函数作用后结果亦应相同，也就是

$$(P \equiv Q) \Rightarrow (F(P) \equiv F(Q))$$

T 是一个隐匿事件,一个自然的观察定义应能导出等价关系

$$(\tau.P) \equiv P$$

但是(τ.P+τ.NIL)却不等价于(P+NIL)；因为后者等同于 P，而前者可能经过非确定性选择导致死锁。

Milner 使用了同余(congruence)的概念来解决这一问题,，通过同余来代替等价(equivalence)。在对进程 P 可以做的各种实验测试中，将进程 P 放在一个环境 F(P)中，研究在环境 F(P)中的试验，此处 F 是由语言中的算子所构成的进程组成的；从而观察整个复合体的行为。若对语言中每个可表示的 F，进程 F(P)是观察等价于进程 F(Q)，则称进程 P 和 Q 是(观察)同余的。按照这一定义,τ.P 和 P 不同余的。发现同余关系的一整套法则是数学上的重要成果。

引入额外复杂的观察同余的原因在于，不将进程 P 放入环境 F(P)中，就不可能对 P 的行为做出足够深入的观察。这也是为什么我们不光引人拒绝事件，还必须引入拒绝集的原因。看来拒绝集是对非确定死锁可能性的最弱的一种观察；因此拒绝集能给出更弱的一种等价关系，从而也给出了比 CCS 更强的代数法则。

上面我们过多地强调了 CSP 和 CCS 的区别，也过多地陈述了本书所采取的途径的实际应用。事实上 CSP 和 CCS 这两个途径的主要特征是相同的，即都提供了形式描述推理，设计和实现的数学基础；这两者都能同时用于理论研究和实际应用。

[278] CSP 理论中的迹(trace)就是可观察的行为。

文献(Bibliogrαphy)

Conway, M. E.

'Design of a Separable Transition-Diagram Compiler,' Comm. ACM 6 (7), 8–15 (1963)

The classical paper on coroutines.

Hoare, C. A. R.

'Monitors: An Operating System Structuring Concept,' Comm. ACM 17 (10), 549–557 (1974) A programming language feature to aid in construction of operating systems.

Hoare, C. A. R.

'Communicating Sequential Processes,' Comm. ACM 21 (8), 666-677 (1978)

A programming language design—an early version of the design propounded in this book.

Milner, R.

A Calculus of Communicating Systems, Lecture Notes in Computer Science 92, Springer Verlag, New York (1980)

A clear mathematical treatment of the general theory of concurrency and synchronization.

Kahn, G.

'The Semantics of a simple language for Parallel Programming,' in Information Processing, 74, North Holland, Amsterdam pp. 471–475 (1984)

An elegant treatment of functional multiprogramming.

Welsh, J. and McKeag, R. M.

Structured System Programming, Prentice–Hall, London, pp. 324 (1980)

An account of PASCAL PLUS and its use in structuring an operating system and compiler.

Filman, R. E. and Friedman, D. P.

Coordinated Computing, Tools and Techniques for Distributed Software, McGraw–Hill pp. 370 (1984)

A useful survey and further bibliography.

Dahl, O–J

'Hierarchical Program Structures,' in Structured Programming, Academic Press, London pp. 175–220 (1982)

An introduction to the influential ideas of SIMULA 67.

(INMOS Ltd.)

occamTM Programming Manual Prentice–Hall International, pp. 100 (1984).

(ANSI/MIL–STD 1815A) AdaTM Reference Manual

Chapter 9 describes the tasking feature.

Brookes, S. D., Hoare, C. A. R., and Roscoe, A. W.

'A Theory of Communicating Sequential Processes,' Journal ACM 31 (7), 560–599 (1984)

An account of the mathematics of nondeterminism processes, but excluding divergences.

Brookes, S. D. and Roscoe, A. W.

'An Improved Failures Model for Communicating Sequential Processes,' in Proceedings NSF–SERC Seminar on Concurrency, Springer Verlag, New York, Lecture Notes in Computer Science (1985)

The improvement is the addition of divergences.

索引(Index)

(This page is left blank intentionally)[279]

[279]译者认为 Hoare 的 CSP 书的 Index 不是很完整和有必要。也有一些错误。不少的索引的页号与书中的出处匹配不上。

勘误表(Errata)

译者在比较阅读 Hoar 的 CSP 2015 电子版(http://www.usingcsp.com/cspbook.pdf。 以下简称 Hoare2015)，1985 年的 CSP 印刷版(Prentice-Hall Internationalf，以下简称 Hoare1985)和周巢尘院士 1990 年的译本《通信顺序进程》(北京大学出版社，以下简称周 1990)的时候，分别发现了一些不一致的地方，或者一些小问题。下面是笔者试图整理的勘误表。这里的"勘误"一词并不意味着确认的对错，更多的是让读者知道上述三个版本略微有不一致的地方。

1. 联立递归 X1：饮料机行为描述(1.1.4)

在 Hoare2015 和 Hoare1985 的书里(第 33 页)，对饮料机的行为描述是一致的，皆为

αDD = αO = αL = {setorange, setlemon, orange, lemon}

DD = (setorange → O | setlemon → L)

O = (orange → O | setlemon → L | setorange → O)

L = (lemon → L | setorange → O | setlemon → L)

但令人迷惑的是，在周 1990 的书中(第 12 页)，饮料机的行为描述有不小的区别如下：

αDD = αG = αW = {setorange, setlemon, coin, orange, lemon}

DD = (setorange → G | setlemon → W)

G =(coin→orange → G | setlemon → W)

W = (coin → lemon → W | setorange → G)

从符号表，到进程行为定义都有区别，例如，周 1990 书里的 G 和 W 的定义。在 Hoare1985 和 Hoare2015 里分别是进程 O 和 L。

2. Traces of process X1 的证明(1.8.1)

Hoare 2015 删除了归纳法证明时，对 n=0 的条件。原因应该是，STOP 进程其实蕴含了 n=0 这个逻辑。

3. istrace 的实现(1.8.2)

Hoare2015 采用了完整的 Lisp 语法来表达，而在 Hoare1985 和周 1990 里，还采用了 CSP 的语法。译者认为 Hoare2015 的修改是正确的。

Hoare2015:

```
istrace(s, P) = if s = NIL then
      true
else if P(car(s)) = "BLEEP then
       false
  else
      istrace(cdr(s),P(car(s)))
```

Hoare1985，周 1990：

```
istrace(s, P) = if s = NIL then
      true
else if P(s₀) = "BLEEP then
      false
else
      istrace(s´,P(s₀))
```

4. Traces L3A(2.3.3)

Hoare2015 的 L3A 里应该是遗漏了 ∃ u。译者认为 Hoare1985 和周 1990 是正确的。

Hoare2015

L3A 如果 αP ∩ αQ = {}

```
traces(P ‖ Q) = { s | ∃ t: traces(P); u: traces(Q) • s
interleaves(t, u)}
```

Hoare1985，周 1990：

L3A 如果 αP ∩ αQ = {}

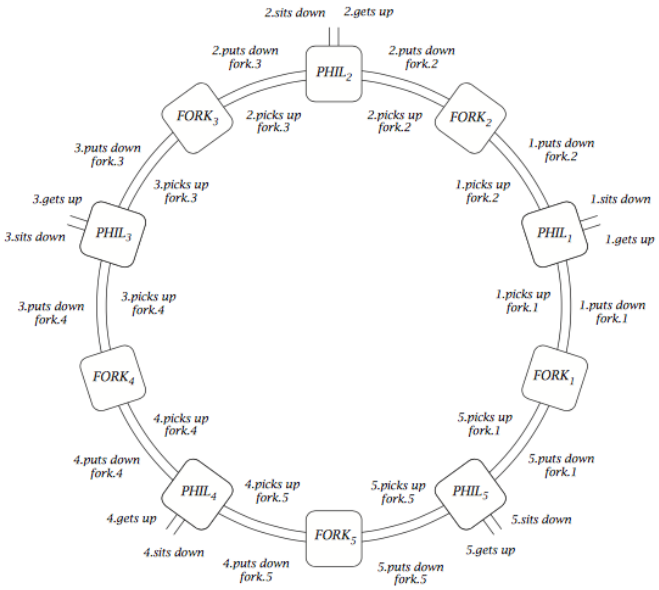
```
traces(P ‖ Q) = { s | ∃ t: traces(P); ∃ u: traces(Q) • s
interleaves(t, u)}
```

5. The Dining Philosophers(2.5)

Hoare2015 图 2.6 中的哲学家和叉子的编号与哲学家和叉子的行为描述里的编号不匹配。例如

PHILOS = (PHIL0 || PHIL1 || PHIL2 || PHIL3 || PHIL4)

FORKS = (FORK0 || FORK1 || FORK2 || FORK3 || FORK4)



但是图 2.6 标注了 PHIL1-5 和 FORK1-5。

Hoare1985，周 1990：

图 2.6 中的哲学家和叉子的编号与哲学家和叉子的行为描述里的编号是一致的。

6. 数学理论基本定义(2.8.1)

在 Hoare2015 和 Hoare1985 书中，对引述的法则标注了一些不能适用的特例。

“为了避免引起混乱，在本书中我们引述的法则不仅可以万无一失地应用于确定性进程，而且完全适用于非确定性进程(除了 2.2.1 L3A，2.23 L1，2.3.1 L3A，2.3.3 L1，L2，L3。这些规则在含有进程 CHAOS(3.8 节)中是不成立的。”

在周 1990 书的相关段落里，没有这个部分。

7. Interleaving(3.6 X1)

在 Hoare2015 中，把 3.6X1 的详细定义去除了。

中 Hoare1985 和周 1990 中，定义如下：

把四个男仆当作一个仆人使用，每次一个男仆只为一位哲学家服务(参看 2.5.3 节及 2.6.4 节中的 X1)

L ||| L ||| L ||| L

其中 L = SHAREDLACKEY

8. 发散(3.8)

Hoare 在 Hoare2015 的 3.8 节中，做了几个意味深长的修正。

- 在讨论 guarded 的模式保障进程有唯一解时，去除了“递归”一词的限制。
- 彻底删除了对 CSP 中重要的概念之一 CHAOS 的定义的完整段落。读者可以参阅 Hoare2015 106 页和 Hoare1985 126 页。依着认为 Hoare2015 可能有错。整个 CSP 书里这个地方是唯一定义 CHAOS 进程的地方。没有 CHAOS 的形式定义，3.8.1 的法则和后续关于发散进程的讨论都缺乏基础。

另外，3.8.1 L8 的定义里出现笔误，漏了"}"字符。

9. 赋值(5.5)

在 Hoare2015 的 5.5 节中， 对 QUOT 的定义出现了笔误如下

QUOT = (q := x + y ; r := x - q × y)

在 Hoare1985 和周 1990 的 5.5 节中，是正确的定义。q 是商(quotient)，应该是

q := x ÷ y

10. 其它

在 Hoare2015 中，还有一些有别于 Hoare1985 的地方，但大多数都是一些小标记的修改，或者笔误例如，5.5.2 X3 去除了一段解释文字。

后序

译者初步学习形式化方法（Formal Method）大概是 1994 年，主要是初步接触了 Tony Hoare 的 CSP 和 Robin Milner 的 CCS。但因为当时的基础比较薄弱，和无人指点，很难往下深入学习。可能是 1994 下半年年或者 1995 年，科学院软件所，时任联合国软件研究院（澳门）院长的周巢尘院士访问译者所在的实验室，并给了一个关于并发系统形式化验证的学术讲座。周院士的谈吐，学术和谦逊的待人接物是令人由衷的钦佩。从此，对符号推理和形式化方法的好奇心也更加深深的埋在了心里。

大概是 1999 年或者 2000 年的一天，译者当时在美国 SRI 研究院[280]从事 DARPA 相关的应用研究工作。突然一个同事敲门说，今天有一个中国来的科学家给一个学术讲座。万分惊喜的发现是周巢尘院士访问 SRI CS Lab[281]。至今还记得在聆听为周院士的讲座后，与周院士交谈的场景。对周院士对晚辈后学的鼓励，至今记忆犹新。

二十年，弹指一挥间。

再次捡起形式化方法和 CSP 是 2014 年。长期埋藏在心里的，对于符号逻辑的好奇心和兴趣，使得艰难的但又执着的，开始了对 CSP 原著的阅读，和对现存的英文和中文版本的再翻译和校注的工作。这个工作断断续续历时四年。许多时候一个貌似简单的逻辑证明可能需要一周或者数周才能理解。加注工作更是异常辛苦，译者试图做到对书中每一个推理，定理都尽力给出了通俗易懂的解释，并对之前的相关遗漏做了一些修订。大约总共加注 280 个。在 CSP 的主要 1-5 章里，加注了 244 个条目。

时光飞逝，又是三年，其中经历了目前还在进行中的 2020 年的 COVID-19。译者再次对对 2018 年的工作进行了整理和修订，并定稿。

---

[280] 是服务于美国 DARPA 和政府的重要的科学研究院之一。1945 年创办于斯坦福大学。1977 年，独立发展，大概有 10 多个研究所做成。涵盖生物，医学，化工和计算等领域。在计算机领域，人工智能研究所和计算机科学研究所是欧美在计算机科学的重要研究基地之一。 计算机领域的重要算法和形式化理论科学家 Leslie Lamport 在 1977-1985 年工作于 SRI CS Lab。互联网（internet），鼠标（mouse）都发源于 SRI。

[281] SRI CS Lab 在形式化方法，逻辑的研究方面，久负盛名，是一个比较重要的研究基地。

这个事情的完成，断断续续，历经 7 年之久，很希望能够在中文文献里，为后来的年轻华人学生/学者，留下一份在符号逻辑推理方面，可能有一点价值的积累。希望这一个非盈利性质的工作是一件对社会有价值，有帮助的事情。

再次深深的感谢 Tony Hoare，Robin Milner（已去世），周巢尘这些令人尊敬的科学家。